ScriptX
Tools Guide

Kaleida Labs

Printed in the USA.

Contents

Part 1 Development Process 3

Chapter 1 Designing a ScriptX Title 5

Titles, Applications and Tools 7

Title Structure 7

The Scenes and Stage Manager Structure 8

The Model-Presentation Structure 9

ScriptX Tips and Traps 9

Tips 9

Traps 12

Speed and Memory Tips 13

Chapter 2 Building a Title from Source Files 15

Title Development Process 17

Source Files 17

Creating a Build File for Source Scripts 18

Creating a Build File for Media 22

Chapter 3 Optimizing for Speed and Memory 25

Optimizing for Speed 27

Speed Tools in ScriptX 27

Language 28

Color 28

Compositing 28

Animation 29

Video 29

Disk Access 30

Optimizing Memory 31

Memory Tools in ScriptX 31

Size of A Class 31

Garbage Collection of Persistent and Transient Objects 32

Rules of Thumb for Creating and Purging Objects 33

Debugging Memory Problems 34

Loading and Purging 35

Presenters and Bitmaps 35

Minimizing The Need For Garbage Collection ... 35
Exceptions can Prevent Garbage Collection ... 36
Visual Memory Tips ... 37
Garbage Collection Tips ... 37
ShowCode Tips ... 37
Other ... 38

Chapter 4 ScriptX Title Analysis API ... 39

Part 2 Kaleida Media Player 45

Chapter 5 Kaleida Media Player User Guide ... 47

Introduction ... 50
Installation Instructions ... 50
Macintosh ... 50
Windows ... 50
OS/2 ... 51
Starting the Title ... 52
Macintosh ... 52
Windows ... 52
OS/2 ... 52
Kaleida Media Player Menus ... 53
The File Menu ... 53
The Edit Menu ... 54
The Window Menu ... 54

Part 3 Tools 55

Chapter 6 Introduction to Tools ... 57

Loading Tools and Importers ... 60
Loading Tools ... 60
Loading Importers ... 61

Chapter 7 ScriptX Listener and Menus ... 63

Features in ScriptX That Are Not in KMP 65
The Development Process 65
Starting ScriptX 66
The ScriptX Listener 67
Typing Scripts Directly into the Listener 68
Creating and Compiling Script Files 70
New Listener Windows 73
Resetting ScriptX 73
The Listener and Other ScriptX Windows 73
ScriptX Menu Command Reference 75
The File Menu 75
The Edit Menu 77
The Window Menu 79
The Tools Menu 79
The *ToolName* Menu 80

Chapter 8 The Browser 81

Loading the Browser 83
Using the Browser 83
Resizing the Browser 84
Stepping Back to the Previous Browser Display 84
Updating Browsers 84
What You See in the Browser 85
Sample Uses 86
Editing Values in the Browser 86
Editing Methods and Functions 86
More On Actual Versus Virtual Instance Variables 87
Using the Picklist 88
Reference 90
Tools Menu 90
Browser Menu 90

Chapter 9 Debugger 93

How to Load The Debugger 95
Opening the Debugger 95
Resizing the Debugger 96
Description of the Debugger 97
Selected Frame Field and Call Stack Menu 97
Frame Variables Panel 98
Code Panel 99
Stepping Through Code 100

Entering the Debugger Immediately 100
Stepping Through Code 100
Setting Breakpoints 103
Defining a Function to Run at a Breakpoint 104
Using Watchpoints 106
Investigating Exceptions 107
Accessing Source Code for Methods and Functions 107
DebugInfo Objects in Title Containers 109
Reference 110
The Debugger Menu 110
Tools Menu 111
The Debugger Window 111
The Watchpoints Window 113
The Breakpoints Window 113

Chapter 10 ByteCodeMethod Profiler 115

How to Install the Profiler 117
How to Run the Profiler 117
What The Profiler Does 118
Profiler Output 118
Example Use of the Profiler 120

Chapter 11 Visual Memory 123

Platform Differences 125
Running Visual Memory in the Kaleida Media Player 126
Details of Visual Memory 126

Chapter 12 Tool Framework 129

Overview 131
ScriptX Tool Support 131
Writing a Tool in ScriptX 132
Implementation of the Tool Framework 133
Tool Startup Process 133
Tool Organizer 134
Tool Communication 134
Tool Messaging 136
Easy Event Handling 138
Tool Host OS Access 139
The DragRegion Class 141
Future Directions 142

Chapter 13 Tool Framework API ... 145

Global Functions ... 147
Global Variables ... 147
ToolOrganizer ... 148
ToolContainer ... 151
  Creating and Initializing a New Instance ... 151
  Instance Methods ... 152
ToolMenuBar ... 155
ToolMenu ... 156
  Instance Variables ... 156
  Instance Methods ... 156
ToolMenuItem ... 157
  Instance Methods ... 158
DragRegion ... 159
  Instance Methods ... 160

Chapter 14 Photoshop Plug-ins for KIC Compression ... 161

What is KIC? ... 163
Where Can the Plug-ins Be Used? ... 164
Installing the Plug-ins ... 164
Using the Plug-ins ... 165
The Playback Side ... 165

Chapter 15 Importing Media ... 167

File Organization for Importers ... 169
Importing ... 170
  Available Output Types ... 170
  Specifying A Container for the Imported Data ... 171
Exporting ... 171
  Text Exporter ... 172
Importers Supplied by Kaleida Labs ... 172
Text Importer ... 174
  ASCII, ScriptX ASCII or RTF Text to Text ... 174
Image importers ... 177
  Pict to Bitmap ... 177
  DIB to Bitmap ... 178
  KIC to Bitmap ... 181
  QuickTime to Bitmap ... 183
Sound importers ... 185
  AIFF to AudioStream ... 185

SND to AudioStream 186
WAVE to AudioStream 188
MIDI Importers..... 190
Standard MIDI to Stream..... 190
Movie Importers..... 192
QuickTime Or AVI to MoviePlayer or InterleavedMoviePlayer..... 192
Batch Processing 196
..... 196

Chapter 16 Using the Director-to- ScorePlayer Importer 197

What the Director-To-ScorePlayer Importer Can Import..... 199
What the Director-to-ScorePlayer Importer Cannot Import 199
Classes and Inheritance..... 200
How Director Titles are Converted to ScriptX Objects 202
Preparing a Director Title for Importing Into ScriptX..... 203
Loading the Director-to-ScorePlayer Importer..... 204
Importing a Director Title as a ScorePlayer..... 204
Using the ImportDirector Function..... 204
Using the Import/Export Engine..... 205
Saving an Imported Animation to a Title Container..... 207
Sample Script that Imports and Saves an Animation in ScriptX..... 208
Playing an Imported Title..... 210
Modifying an Imported Animation..... 211
How an ActionList Player Works 211
What is the Target of An Action in an Animation?..... 212
Handy Hints for Keeping Control of Targets..... 213
The Classes of Action..... 213
How to Read the Action List of a Score..... 217
Retrieving Information about an Animation 220

Chapter 17 Director-to-ScorePlayer Importer API 221

AddSpriteToStageAction 224
ChangeInkAction 226
ChangeSoundAction 227
ChangeSpriteAppearanceAction 229
ChangeSpriteSizeAction 230
ChangeTempoAction 231
DirectorImporter 232
LoopAction 233
PrepareTransitionAction 234

RemoveSpriteFromStageAction class 236
Score 237
ScorePlayer 239
ScorePlayerCastTranslator 242
ScorePlayerScoreTranslator 244
ScoreTicker 245
Sprite 246
SpriteChannelInfo 247
TextSprite 248
TwoDSprite 250
VideoSprite 252
WaitForSoundAction 253

Chapter 18 Using The Director Translation Kit 255

Classes and Inheritance... 257
How the Director Translation Process Works.. 260
Cast List Translation... 261
Score Translation 262
What You Must Do To Build a Customized Director Importer 262
1. Design a Paradigm for Recreating a Director Title in ScriptX................. 262
2. Load the Director Translation Kit 262
3. Define Your Customized Cast Translator Class 263
4. Define Your Customized Score Translator Class 263
5. Import a Director Animation into ScriptX 264
Designing a Paradigm for Recreating a Director Title in ScriptX....................... 264
Defining Customized Cast Translator Classes... 265
Choosing Superclasses for Your New Cast Translator Class 265
Accessing Information about the Director Castmembers 268
Defining Customized Score Translator Classes... 268
Accessing Information about a Frame of the Director Score...................... 269
Comparing Two Frames to Check for Differences 270
How the Score Translator Gets Information about Castmembers.............. 270
Example Code for a Score Translator 271
Importing a Director Animation into ScriptX 274
Advanced Example 277
Preparing the Director Title... 277
How the Conversion Works 278
Further Modifying the Example... 279

Chapter 19 Director Translation Kit API 281

DTK 284

DTKBitmap 286
DTKButton 287
DTKCastMember 288
DTKCastMemberToAudioStream 289
DTKCastMemberToPresenter 290
DTKCastMemberToStencil 292
DTKCastTranslator 294
DTKPalette 296
DTKScoreFrame 297
DTKScoreTranslator 298
DTKShape 300
DTKSound 302
DTKSoundChannel 303
DTKSpriteChannel 304
DTKTempoChannel 306
DTKText 307
DTKTransitionChannel 309
DTKUnknown 310
DTKVideo 311

Part 4 Extending ScriptX 313

Chapter 20 Extending ScriptX 315

Using the Loader to Run Loadable Extensions 317
Loadable Units 320
Invoking the Loader 320
Releasing Loadable Units 322
Writing Loadable Extensions 322
Combining ScriptX and External Code. 322
Creating a Loadable Unit 322
Examples of Loadable Extensions 324
API for ScriptX Loadable Extensions 333
SXextend.h Header File 343
C-Language Development Environments 345
Apple Macintosh 68XXX 345
Apple Macintosh PowerPC 345
Microsoft Windows 346
OS/2 347

Chapter 21 Inter-Application Communication 349

Dynamic Data Exchange (DDE) 351

Apple Events 351

Apple Event “Do Script” 351

Index 353

Preface

This document is part of the ScriptX Technical Reference Series. This series is for programmers using ScriptX to develop interactive multimedia tools and titles. This series includes the following documents:

- The *ScriptX Components Guide* provides an overview of ScriptX architecture, conceptual explanations about the organization of the ScriptX classes into components, and script examples showing how the classes work together. It covers ScriptX from the multimedia title, down to the operating system devices. This manual is essential to anyone designing and building multimedia titles in ScriptX. It is the companion volume to the *ScriptX Class Reference*.
- The *ScriptX Class Reference* is a detailed reference to the ScriptX class library that provides, in dictionary form, a complete specification of the classes, methods, variables, and functions available for building multimedia titles and tools in ScriptX. It is the companion volume to the *ScriptX Components Guide*.
- The *ScriptX Language Guide* is a practical guide to using the ScriptX programming language. It provides complete functional descriptions of the language as well as concrete descriptions of tasks you might do when actually working with the ScriptX language. Anyone programming in ScriptX will want to use this book.
- The *ScriptX Tools Guide* (this manual) provides information about the ScriptX development process that is not covered in the other manuals. The first part discusses how to use the browsers, the Listener and other tools that are supplied with ScriptX. All users will want to read this part. The second part explains how to extend ScriptX by loading classes written in C, and discusses platform-specific issues. Developers who wish to add classes written in C to ScriptX will want to read the second part. The third part of the *ScriptX Tools Guide* discusses how to build additional tools in ScriptX. Tool developers will want to read the third part.
- The *ScriptX Quick Reference* summarizes information about the ScriptX Language and Class Library. It includes the grammar of the language, listings of components and their classes, and an alphabetical reference to classes, including class variables, instance variables, and methods.

Summary of Contents

- Part 1, “ScriptX Development Process,” describes the development process.
- Part 2, “Kaleida Media Player,” describes the runtime version of ScriptX from the end-user’s perspective.
- Part 3, “Tools,” describes how to use the listener window,, debugger, browser, profiler, visual memory, importers, Director Tool Kit, and other tools that are supplied with ScriptX.
- Part 4, “Extending ScriptX,” explains how to extend ScriptX by writing and loading C classes, and describes the platform-dependent development environments.

Who Should Read This Book

All ScriptX developers should read the first part. Developers who wish to add classes written in C to ScriptX should read the last part. Tool developers should read the third part.

This manual assumes the reader is familiar with the ScriptX language and object-oriented programming concepts. See the *ScriptX Language Guide*, the *ScriptX Class Reference*, and the *ScriptX Components Guide* for more information on ScriptX programming.

Manual Conventions

This manual is set primarily in Palatino and ITC Avant Garde, with the following exceptions:

- Code samples, class names, method names, and other code-like elements are in `Courier.`
- Menu commands are in **Palatino Bold**.

Note – Notes to the user look like this.

PART ONE

Development Process

CHAPTER

1

Designing a ScriptX Title

This document explores some key issues of ScriptX title design. It describes techniques to help you take advantage of ScriptX features while avoiding pitfalls that adversely affect your title's size and performance.

The discussion that follows is based on the observations of numerous ScriptX developers within Kaleida. It represents "current thinking" on ScriptX title design. Some of the techniques discussed represent alternative—and possibly mutually exclusive—approaches to the same issue. They are presented to help you choose the approach that works best for your particular title.

Some of the design techniques discussed in this chapter are demonstrated in examples: *Auto Finder* is available with the current ScriptX CD-ROM. *Monterey Canyon* is available on the previous version, ScriptX Language Kit V1.0 CD-ROM. These examples are used in this document to demonstrate ways to organize the development and the structure of a title. Other examples on the ScriptX release demonstrate different approaches to the same tasks. You may find you gain the most by first reading through this chapter to understand ScriptX design fundamentals, then reviewing the other examples to see how they approach title design issues.

Note-This description assumes you understand the ScriptX Language and Class Library. While some of the topics may be useful to a beginner, they're more likely to be helpful after you've spent some time reading other documentation and working with ScriptX.

Finally, note that much of this discussion is based on techniques that have been tried and proven to work. Given the short history of ScriptX, it's likely that many other techniques can be used to improve title design; hopefully, this document will give you a basis for working on alternate approaches.

Titles, Applications and Tools

When starting a design, one of the first things you will be aware of is whether you are creating a title, an application or a tool. Here are loose definitions of these terms based on how they are used:

- Title – in a title, the end-user navigates and interacts with content
- Application – in an application, the end-user produces and manipulates content
- Tool – in a tool, a developer produces content, titles and applications, and other tools

These distinctions can be blurred, as titles become more and more interactive.

Title Structure

The abstract structure of a ScriptX title is provided through the models-presenters-controllers system described in the "Spaces and Presenters" chapter in the *ScriptX Architecture and Components Guide*. This structure is metaphor-free and provides a high degree of flexibility in title development. The examples *Beaker and Flame* and *Play Farm* (from the ScriptX V1.0 CD-ROM) demonstrate the use of separate modeling and presentation spaces.

However, a variety of titles don’t require the model and presentation to be separate. In some cases, separating the two can lead to over-abstraction and unnecessary indirection, resulting in slower performance. Titles where the presentation and model are not separated, that is the presentation is the model, are illustrated in *Monterey Canyon* and *Auto Finder*. These two example titles are structured around scenes.

Note that there are many varying ways to structure a title. The next section describes how you might structure a title that is arranged in scenes.

The Scenes and Stage Manager Structure

When a title allows the user to move from one scene to another, it’s natural to design the title to be structured in scenes, with a stage manager that oversees the transitions between scenes. Each scene has different media, so that the transition requires unloading media for the previous scene and loading the next one.

Both the *Monterey Canyon* and *Auto Finder* titles define a structure organized into scenes and a stage manager.

Scenes

Each scene is a complete, contained subset of the user presentation. Scenes are defined as `TwoDSpace` subclasses whose methods correspond to behavior within a scene. Each scene knows about the media it presents and defines the user interaction it provides. For example, the `Rover` class in *Monterey Canyon* defines the rover scene that lets you explore the underwater canyon.

In the code for various scene classes, instance variables are used to represent the state of user interaction and subpresenters for the media presented in the scene. Instance methods perform several types of action:

- Load media for the scene from the title’s media container and prepare subpresenters for the scene.
- Implement user interaction with the scene.
- Provide memory management for the scene, to ensure that when it is finished, all objects it manages are dereferenced so they can be garbage collected.

The Stage Manager

The stage manager acts as the manager for the title as a whole, overseeing the transitions from scene to scene.

The `TitleContainer` class provides a useful starting place from which to implement top-level management of a performance. Both *Monterey Canyon* and *Auto Finder* define a subclass of `TitleContainer` that provides stage management. Here’s a part of the class definition for the `MontereyCanyonTitle` class:

```
class MontereyCanyonTitle (TitleContainer)
instance variables
    stage
    currentScene
    stopButton
    media
    mouse
end
```

In these instance variables, `stage` represents a `FullScreenWindow` where each scene is presented, `currentScene` is the scene object currently being presented, and `media` is the library container for the media belonging to the current scene.

Rather than storing objects representing each scene, the stage managers for both *Monterey Canyon* and *Auto Finder* store classes representing each scene. *Auto Finder*'s technique for storing these classes is shown in the `BUILDME.SX` script in "Building a Saved Title" on page 19. The stage manager doesn't create instances of these classes until the scene is ready to be performed.

As the title runs, the stage manager's main task is to provide transition from scene to scene. To do so, it purges the previous scene, if any, then creates the instance representing the next scene. In version 1.0 of ScriptX, this technique has proven to be more reliable than creating and storing instances of each scene class in the title container.

The Model-Presentation Structure

The model-presentation structure is another type of structure in which the model and presentation are represented by different sets of objects. Objects in the model are directly mapped to objects in the presentation, so that a change in the model changes the presentation, and vice-versa.

This structure is used by the *Beaker and Flame* example, which has a model space for the underlying, non-visible thermodynamic model and a separate space for presenting the model to the user. The *Play Farm* title from ScriptX V1.0 also takes advantage of the separation of model and presentation.

In using this structure, the model is one space and the presentation is a window (another space). You should create a new class for each new type of space created. Each space should manage the objects and media that make up its behavior. For example, each space should bring in and purge the objects, as needed, that belong to it. The structure of such a title can be similar to that described in the Scene-Stage Manager description, with the Stage Manager managing both scenes and models.

ScriptX Tips and Traps

Here are some general title design pointers in no particular order.

Tips

Import and store media into library containers

Use a keyed collection as the target for the library container; this lets you refer to the media by named rather than by integer position in the default array.. Create a script to build these media containers. As you import and store media, keep an eye on visual memory. This lets you preview the sizes of various media resources.

Use a strategy for incremental title modification

While authoring, you can create part of a title by loading previously-compiled storage containers, and create the rest of the title by compiling the code you are currently working on. (At playback, titles are loaded strictly from storage containers.) To add developer modifications, you can incrementally update titles in several ways: through separate loadme and buildme scripts, with patch files, and by compiling free methods in the ScriptX Listener.

Structure scene-oriented titles in a scene-oriented way

Use the scene-stage manager framework where appropriate. Use scenes to manage media, user interaction, and memory usage. Make sure each scene cleans up all its object references or calls `requestPurge` appropriately before it closes. Define a stage manager to coordinate the transition from scene to scene in a title.

Be aware of thread contention

Threads allow multiple concurrent execution of scripts. They're useful in executing concurrent activities, such as doing a database search while an animation runs in the main thread. As noted earlier, they're useful for complex event handler functions, where event handling might otherwise overrun the event handler thread stack. However, the overhead associated with running multiple threads can create some constraints. If a title tries to run too many threads at once, none gets sufficient time and overall performance suffers.

Use callbacks for timed behavior

Use callbacks rather than threads for repetitive time-sensitive actions. Unlike threads, periodic callbacks can be set to skip when system performance is bogging down.

Collections

Collections are one of the more pervasive features of ScriptX, and can be used in a variety of contexts. For example, you can combine a group of related objects into a single keyed collection, then assign the collection to a global variable. This provides global access to every object in the collection with the use of only one global variable. This is the role played by `theTitleContainer` global variable.

You can also use collections to perform higher-order programming within a title. For example, you can put objects of any type into a collection, including classes, functions, and generics. With such collections, the user can create new instances, call specific methods on selected objects, and perform other types of "runtime programming."

Be aware of the memory overhead of modules

ScriptX modules provide a useful mechanism for preventing name collisions. However, each module you define has an associated memory overhead that you should be aware of. Use Visual Memory when loading modules to see their affect on available memory.

Be aware of the size of your media

Common sense applies: large bitmaps and other media take up big chunks of memory when they're loaded. Use the Visual Memory tool both when importing media and as a scene loads to see how much space is allocated. Pare down bitmaps using techniques described in the next two tips.

Use compressed bitmaps for static and background images

Compressed bitmaps allow you to trade a slight hit in performance (when transitioning into and out of a scene) for a smaller memory footprint. See the QuickTime to Bitmap and DIB to Bitmap importers in the *ScriptX Developer's Guide*for details on importing compressed bitmaps.

Crop Your Bitmaps

You can reduce the memory required for bitmaps by reducing them to the minimum bounding box of non-invisible pixels, and translating the bitmap appropriately to compensate. This technique was used with the *Play Farm* example (from ScriptX V1.0) and resulted in a 50% reduction in media storage requirements for the title.

The method `shrinkBitmap` is included in the `MediaImporter` class defined in *Monterey Canyon*. Using this method makes importing take longer, but it cuts down on storage size, playback size, and the time required by the compositor, since there's less area to update.

The code in `shrinkBitmap` makes a few assumptions—for example, it assumes that the upper left pixel is the `invisibleColor`, and the bit depth is eight.

Use invisibleColor for bitmap animations

Compared to `matteColor`, using the `invisibleColor` of a bitmap provides quicker load times and animation (these two `Bitmap` instance variables are mutually exclusive). With `matteColor` set, the 2D Graphics system determines transparency by testing whether a pixel of the specified color is within or outside the image area. With the `invisibleColor` set, all pixels of the specified color are simply treated as transparent.

Methods vs. Functions (Speed Tip)

In general, define classes and methods when you find yourself writing more than one function related to a particular area of functionality within a title. Use functions in performance-critical areas such as callbacks and high-speed loops (functions do not have the overhead of a generic dispatch). The titles *Monterey Canyon* and *Auto Finder* both demonstrate good object-oriented coding, with subclassing limited mostly to the classes for title and scene management, and wide use of the classes defined in ScriptX for most other purposes.

Make sure objects are freed when no longer needed

Here are some pointers:

- Empty out windows after hiding them to make sure they will be freed.
- Cancel all callbacks on a clock to make sure it will be freed.
- If a player has had its `playPrepare` method called, call `playUnprepare` to free buffers and perform other cleanup.
- Keep track of places where you're creating references to objects. Collections hold onto the objects that you add to them; instance variables hold onto objects assigned to them. For unsaved objects, be sure to `emptyOut` collections and set instance variable references to `undefined`; for saved objects, call `makePurgeable` on them.
- Remove event interests; event interests may hang onto other objects.
- Only if necessary, to ensure complete collection, call the `garbageCollect()` function.

Global Variables

Avoid defining global variables within a title, since, when used in a method, restricts the way that method can used by other instances. In general, global variables set you up for memory management problems and impede code maintenance.

You can avoid declaring globals within a title by smart use of system-defined global variables, class variables, and instance variables. For example, `theTitleContainer` global gives you access to the container that currently has focus. Be aware that multiple titles may be running simultaneously, and (particularly with threads) your title's code may be executing when another title has focus.

The following global variables can be very useful :

`theStartDir` The directory where the ScriptX or Kaleida Media Player (KMP) executable resides.

`theScriptDir` (Not in KMP) The directory that the most recently compiled (or currently compiling) script started up from. The value is set by the `fileIn` method of `DirRep`. This variable allows a script to create a title container in the same directory as its build script. It also lets you refer to other files in the directory where the script resides, or to create directory paths relative to that directory.

`theRootDir` The root directory of the file system of the platform on which ScriptX or the KMP is running.

`theTitleContainer` The title container that currently has user focus. Note that the instance variable `theTitleContainer.directory` gives you access to the directory where the current title container resides.

`theTitleContainer.windows[1]`

The window that currently has user focus.

Also note that it's not good practice to change instance variables of the system-defined global constants. For example, the 2D Graphics component defines `redColor` as an instance of `RGBColor`. While you can change the individual `red`, `green`, and `blue` instance variables of this color, doing so obviously defeats its purpose. (The instance `redColor` is considered a constant because you cannot assign another instance of `RGBColor` to it.)

Save instances vs. saving classes in your title

If your title needs many, small, short-lived objects, you can save memory by not storing the instances, but instead storing the class and instantiating them only at runtime. This is because, while the body of an object can be freed by calling `makePurgeable` on it, the handle of an object, once loaded from storage, cannot be freed from memory (unless you make the object not reachable).

Traps

Don't use different colormaps for a bitmap and its window

This causes color remapping on every frame, which is a common but hard to find performance killer. All bitmaps in any window should have the same colormap. Call the `warnings` function to be notified of this color remapping.

```
warnings true
```

This function enables a warning that appears in the Listener informing you each time color remapping occurs in your title.

Avoid Needless Specializing (to Save Memory)

ScriptX provides a rich core class library that can often suffice for many types of behavior. Be careful to avoid needless specializing of classes and objects. Each new class, including the hidden class created when you customize an object, takes up memory that can't be reclaimed, even if there are no instances of that classes.

Event Handlers

The event handling thread, `EventDispatchQueue`, has a limited stack depth. Thus, if you write an event handler function that calls another function that calls another function and so on, you may overrun the stack and cause events to get lost. You can avoid this by keeping the code in your event handler functions simple, or by writing event handler functions that create their own threads and return, thereby allowing the event handler thread to continue processing events.

Speed and Memory Tips

For tips specific to speed and memory, refer to the chapter "Optimizing for Speed and Memory" in this book.

Building a Title from Source Files

2

This document describes how to create build files which load the text and media files for a title.

The discussion that follows is based on the observations of numerous ScriptX developers within Kaleida. It represents several alternatives for ScriptX title building. Much of this description is based on techniques that have been tried and proven to work. Given the short history of ScriptX, it's likely that these techniques can be refined. Hopefully, this document will give you a basis for working on alternate approaches.

Note-This description assumes you understand both the ScriptX Language and the Class Library. While some of the topics may be useful to a beginner, they're more likely to be helpful after you've spent some time reading other documentation and working with ScriptX.

Title Development Process

A general approach to the title development can be summarized with the following steps:

1. Write and compile the scripts that produce the skeleton for your title
2. Create the media in other applications (text, images, sound, animation, video, audio)
3. Write and compile a build script to import and store that media in a library container
4. Write and compile scripts that use that media and complete the title
5. Test and debug the scripts
6. Write and compile a build script that creates a title container and files-in your scripts
7. Test and debug the built title container
8. Test the title container in the Kaleida Media Player
9. Test the title container on other platforms (Windows, Macintosh, Power Mac, OS/2)

Throughout title development, you may will need to repeat some of these steps. That is, you may need to initially import media, then create and compile scripts that use the media, then refine and reimport the media, and so on.

Creating build files, steps 3 and 6, are described in further detail in this chapter.

Source Files

When you build a title, you generally start with two different kinds of source files:

- Script files containing ScriptX scripts in ASCII text format (extension: SX)
- Raw media files that hold the text, images, video, animation and sound (extensions such as: RTF, DIB, AIF, SND, WAV)

To build a title, you compile the script files, some of which can import the media. While you could import media files on-the-fly from the KMP at runtime (if you include the importers), importing media can often be a time-consuming process. Therefore, in general you build a separate script just to import the media into its own library container. This process is described next.

A file that compiles source files we call a “build” file. It builds all or part of the title. The following sections describe how to create a build file for scripts and for media

IYou can develop a title in stages. using both objects from open storage containers, and from freshly compiled scripts. You can incrementally update the scripts and compile them at any time as you refine and debug your title.

In its deliverable form, your title will consist entirely of compiled code—that is, code in storage containers. The Kaleida Media Player doesn’t include the ScriptX bytecode compiler, so all the objects it loads must already be compiled. (The KMP can import media into existing objects if you include the importers when the user installs it.).

Creating a Build File for Source Scripts

Both *Monterey Canyon* and *Auto Finder* (included with the earlier version ScriptX V1.0) have separate scripts for building both saved and unsaved versions of a title. The advantage of the unsaved version is that it is faster to run—you don’t have to close and re-open the title container as you do with the saved version.

Building an Unsaved Title

Authoring Title Loader

The authoring version of the *Auto Finder* title is created by the script in the file LOADME.SX. This script loads and compiles class definitions for the title. Here’s the code for *Auto Finder*’s LOADME.SX script.

```
--<<<-
-- Filename:
-- loadme.sx

-- Other Files Required:
-- All those specified in a fileIn.

-- Purpose:
-- Compile and execute the Auto Finder title from scripts.

-- Specialized Classes:
-- None

-- Instructions to User:
-- File this in, then execute the title from scripts.
-- Note that the menu bar will not be hidden when running from
-- scripts, so that the menu can be accessed when debugging.

-- Author:
-- Steve Mayer

fileIn theScriptDir name:”function.sx”
fileIn theScriptDir name:”crit.sx”
-- ... load a lot of scripts ...
filein theScriptDir name:”adlayout.sx”
```

```
fileIn theScriptDir name:”autofind.sx”

global af
if (not isDefined BuildOnly) do
(
   af := new AutoFinderTitle -- should be a path: keyword here
)
“Compiled loadme.sx”
-->>>
```

After loading all the script files that define classes, the `LOADME.SX` script conditionally creates a new instance of the title’s manager class `AutoFinderTitle`. `AutoFinderTitle` is a subclass of `TitleContainer`, used to both store the title and provide high level management of title-specific behavior.

Note-The `LOADME.SX` script for *Auto Finder* on the ScriptX Version 1.0 CD-ROM generates an exception. The line commented above should include a `path:` keyword and argument.

For authoring purposes, the title only needs to be managed, not stored. When you compile the `LOADME.SX` script directly, the conditional global variable `BuildOnly` is undefined. The script therefore creates an instance of `AutoFinderTitle`, but no objects are saved to it. The instance exists only within the current ScriptX session so the author can run its startup action. The startup action is run by calling:

```
start af
```

where `start` is a method defined in the *Auto Finder* title container `af` to call `startupAction`.

Note-The normal behavior of a title container is to create a new file when it is first instantiated. This file should be deleted after each time the `loadme.sx` script is run.

Building a Saved Title

To create the a version of the title that is saved to atitle container, *Auto Finder* uses a second script, `BUILDME.SX`. The `BUILDME.SX` script defines the `BuildOnly` global, then runs the `LOADME.SX` script. With `BuildOnly` defined, the `LOADME.SX` script doesn’t create a manager; it simply loads and compiles all the scripts defining the classes. Then the `BUILDME.SX` script takes over, creating a file-based instance of `AutoFinderTitle` named `tc`. This container is used to save the title for playback in the KMP.

```
--<<<-
-- Filename:
-- autofind.sx

-- Other Files Required:
-- loadme.sx

-- Purpose:
-- Builds the Auto Finder title container.

-- Specialized Classes:
-- AutoFinderTitle
-- Instructions to User:
-- Run this script to build the Auto Finder title container.
-- It compiles all of the Auto Finder classes (by filing in
-- loadme.sx), and then builds a list of all classes and puts
-- them in the TitleContainer. It defines the start up action to
-- load the classes and create an instance of AutoFinderTitle.

-- Author:
-- Steve Mayer

global BuildOnly := true

fileIn theScriptDir name:"loadme.sx" quiet:true
global titleContainerFilename:= "autofind.sxt"

-- Function sxSubs() finds all scripter defined classes.
function sxSubs ->
(
    chooseAll (getSubs RootObject) \
    (v arg -> sxDefined v) 0
)

-- Create a title container and write its script to load
-- classes and run the title.
global tc := new AutoFinderTitle dir:(parentDir theScriptDir) \
        path:titleContainerFilename
tc.startUpAction :=
(tc ->
    (
        hide tc.systemMenuBar
        -- Load in the classes
        forEach tc load undefined
        start tc
    )
)

-- Get a list of all scripter defined classes, and store them in a
storage container.
global classList := sxSubs()

-- store all the classes in the title container
for cl in classList do
(
    append tc cl
)
append tc setters
append tc getters
append tc fade

close tc
-->>>
```

Notice the function sxSubs. This function finds all the scripted classes and saves them to the title container tc. Notice also the title startup action which loads all the contents of the container tc, then starts the title. After appending all the classes of the title in the container, the BUILDME.SX script closes the container, thus saving the title. (In *Monterey Canyon*, a separate script, BLDCLS.SX, is run to add the classes to the title container.)

Note-Some developers suggest that you avoid using the sxSubs function. Instead, they suggest you explicitly track the classes defined in your title. This is because sxSubs will return a collection of all scripted classes, including those used to implement the importers, browsers, and debugger. The other way to avoid this side effect is to run the BUILDME.SX script only after restarting ScriptX with no tools installed.

Using this approach, code maintenance becomes somewhat more modular. If new source files are defined, they only need to be added to the LOADME.SX script—BUILDME.SX picks them up automatically when it compiles the LOADME.SX file.

Incremental Development Using Free Methods

You can modify a compiled class directly in ScriptX by using free method definitions:

```
method foo {class foobar} -> (
....redefine method code here....
)
```

Enter new code for a method into the Listener to change the behavior of any class for the duration of the current ScriptX session. You can repeatedly redefine the methods of any existing classes in ScriptX, even those that already have defined instances.

Incremental Development Using a Patch File

A further refinement of the previous technique for developing a title is to create a startup function (assigned to a title container’s startupAction instance variable) that loads a patch file that contains only free method and function definitions. This patch file provides a place to add incremental modifications to a title. (It’s difficult to use this technique when you need to modify the init method for a class.)

The startup function is defined to start the title in three steps:

1. Load classes from the title container
2. Load and compile the patch file using the fileIn method.
3. Invoke the method or function that actually starts the title.

The code in the patch file consists of new and modified definitions of methods for the classes in the container. All methods defined in the patch file use the free method construct (shown above):

Code in the patch file can be used to modify or add methods to existing classes, and to define new classes. You can modify methods by changing their behavior, but not by changing the number of arguments. To modify a method’s argument signature or to add or remove instance variables, you need to make the changes directly to the class source file, then once again compile and store the class. You can use a file like loadme.sx and buildme.sx described in the previous section to compile and save the classes. As you develop and test code in the patch file, you can move tested methods into the original source for your classes, then rebuild the title container.

Note that you only want to load the patch file if it exists, and if it can be compiled. The following code tests to see if the compiling method fileIn is defined (it is in ScriptX and isn’t in the Kaleida Media Player), and if there’s a patch file in the source directory:

```
if ((isDefined fileinI and (isthere tc.directory "SOURCE/patch.sx")) do
      fileIn (spawn tc.directory "SOURCE") name:"patch.sx"
```

This code would be placed in the startup function for a `TitleContainer`; it can be left in even when the container is built for playback on the KMP.

Note-You may also want to design your title so that setting a global variable within the patch file directs your title to start with the particular scene you're working on.

Creating a Build File for Media

This section describes how to create scripts that import the media of a title and place it in library containers. As you develop your title and its media assets, you can update these scripts incrementally to reflect changes.

We'll use the *Monterey Canyon* example to demonstrate how to create a build file for media.

Monterey Canyon Example

Monterey Canyon uses two scripts for importing its media: `mediaimp.sx` and `buildme.sx`. These scripts are in files in `dsgnexmp/monterey/media` on the previous release of ScriptX, the V1.0 CD-ROM.

Note-The *Monterey Canyon* example is not included with the current version of ScriptX. It was available in the previous release, V1.0.

The MediaImporter Class

The `MediaImporter` class (defined in the file `mediaimp.sx`) manages the import and storage of all the media types used in the title. `MediaImporter` defines methods to import various types of media and store them in library containers. It defines instance variables to keep track of the container where media is being saved, the type of media being imported, and other information. It also defines a method for shrink wrapping (cropping) bitmaps as they are loaded, by eliminating rows or columns of surrounding pixels that are made up entirely of invisible color.

The Build Media Script

The file `buildme.sx` contains a script that tracks media resources for *Monterey Canyon*, and uses a `MediaImporter` instance to import and store that media in a new library container. The first part of that script is shown here.

```
--<<<
fileIn theScriptDir name:"mediaimp.sx" quiet:true

-- Put resulting title container in monterey folder.
global montereyDir := parentDir theScriptDir
global l := new LibraryContainer dir:montereyDir \
   targetCollection:(new HashTable) path:"monterey.sxl"
global m := new MediaImporter dir:theScriptDir container:l

-- ******* INTRO MEDIA *******
m.convertToShapes:= false
m.invisiblecolor:= whitecolor
importDIB m "rover01.bmp" mediaKey: "rover1"
importDIB m "rover02.bmp" mediaKey: "rover2"
importDIB m "rover03.bmp" mediaKey: "rover3"
importDIB m "rover04.bmp" mediaKey: "rover4"
importDIB m "rover05.bmp" mediaKey: "rover5"
m.convertToShapes:= true
m.invisiblecolor:= undefined
importDIB m "kaleida.bmp" mediaKey: "kaleida logo"
importDIB m "introsea.bmp" mediaKey: "intro sea"
importDIB m "mountain.bmp" mediaKey: "mountain"
...
```

Note that this script uses the `MediaImporter` instance `m` to import the various types of media scene by scene. It places the media for all scenes in a single library container, `l`. In contrast, *Auto Finder* stores the media for each scene in a separate library container.

Both techniques have benefits. Using one media container for the whole title reduces the number of files that makeup a title, but takes longer to save. One the other hand, using one media container for each scene allows media for that scene to be developed separately, perhaps by different developers. Another technique would be to start out using several media containers, then build the final version to a single media container.

In both *Monterey Canyon* and *Auto Finder*, the target collection used by each media container is a keyed collection (a `HashTable` instance in Monterey). Each piece of media is identified by a particular named key, rather than by an index number (which you would get with the default `Array` for the title container's target collection). This makes it easier to rewrite and recompile scene scripts and rebuild media containers independently, without disturbing the scene's ability to access its media.

CHAPTER

3

Optimizing for Speed and Memory

There is an inherent tradeoff between reducing the memory requirements of a title or application and maximizing its speed. In a world with no memory limits, you could maximize speed by loading the entire title into memory. In the real world, you can instead pre-load from disk into memory just those objects that will potentially be used next.

For information on the amount of memory required to run ScriptX, refer to the installation section of the *ScriptX Quick Start Guide*. For general concepts on how memory allocation works internally, refer to the chapter "Memory Management" in the *ScriptX Components Guide*.

This chapter gives many different options for how to optimize your title. Typically, optimizing is a matter of trading off speed and memory. If one technique is faster, but another takes less memory, and you're not sure which one to use, try the one that takes less memory first and see if it's fast enough to do the job. If it is, you're home free; if not, try the faster technique.

This chapter has two sections:

- Optimizing for Speed
- Optimizing for Memory
- Reducing File Size

Important – During development, you should constantly measure speed and memory to catch and correct unfavorable changes when they occur. Developing an entire title and then optimizing it at the end is needlessly painful and time-consuming, and you may discover too late that you are missing your target memory size.

Optimizing for Speed

In general, constantly monitor the speed to catch slowdowns. Add timers that log intervals. Pre-load objects wherever possible.

Speed Tools in ScriptX

- Bytecode Profiler
- Functions to optimize speed. These are documented in the "Title Analysis API" chapter in this document.

  `fps()` – (frames per second) (core function)

  `ptt()` – (print thread timing) (core function) `rtt()` – (reset thread timing) (core function)

`warnings true` – (core function)

When warnings are turned on, the `fps` function will print out the number of frames the compositor was idle (if non-zero). This will help determine if the frame rate is slow due to problems other than drawing speed. This `fps` printout helps developers optimize their titles for performance. The printout can help a developer determine why they are getting a low frame rate. If the compositor isn't idle then they are just trying to do too much. If the compositor is idle, then the bottleneck is artificial.

Language

- Minimize repeated generic access to instance variables. If you need to get the value from a variable more than once, set it to a local variable and use the variable instead. For example:

  ```
  local tc := tree.color
  ```

  use `tc` rather than using `tree.color` numerous times. In ScriptX, setting or getting an instance variable is a call to a generic (rather than an offset). The `tree.color` in the previous example is equivalent to:

  ```
  colorGetter tree
  ```

  However, be aware that this does not work for *setting* a variable value. Setting `tc` does not set `tree.color`.
- A function call is faster than a method dispatch (this is small effect and might not be noticeable except in tight loops). Using a function call is not object-oriented. Consider using this where appropriate—to eek out a tiny bit of speed improvement when a generic would otherwise have just one method.

Color

- A bitmap should use the same color palette as its window. This is very important for speed.
- `matteColor` is slower than `invisibleColor` (does flood fill) use invisibleColor then where you don't want it to be transparent, change the color slightly

Compositing

- Disable the compositor when arranging presenters (set its `enabled` instance variable to `false`).
- Use direct presenters when appropriate, such as for video (set the `direct` instance variable to `true`).

- Make sure `notifyChanged` gets called only once during a composite cycle for each presenter that needs to be reupdated.
- Mark presenters as stationary and non-transparent whenever applicable.
- Minimize the number of presenters that are changing simultaneously.
- A new compositor instance variable `useOffscreen` defaults to `true`, but when set to `false`, all presenters go “direct” to screen, bypassing the off-screen buffer. This saves a whole chunk of memory, and is actually quicker. The disadvantage is flicker in both animation and overlapping stencils. Setting `useOffscreen` to `false` is also useful for debugging and understanding how the compositor works and then rearranging your title for optimal speed.

Animation

- Write the code that runs animation sequences and loops so that it avoids generating needless garbage. Often code can be written so that you reuse the same object over-and-over again, rather than creating new objects each loop. The garbage collector can take up 15% overhead.

Video

- Call `playPrepare` on video and audio players to minimize switch time. To reduce seek time on a sequence of videos from off the CD. We suggest calling `playPrepare` in reverse order. (Due to lack of asynchronous reads on Microsoft Windows, some seek time is unavoidable there.)
- When switching video clips, you have a choice between storing them as separate clips, each with their own player, or storing them all in one long movie, with programmatic control over what section of the movie is playing. The segmented movie seemed to be the most efficient approach.

  One problem to be aware of when using segmented movies:

  - You don't have predictable control of a movie player on a frame-by-frame basis. Depending on what else is going on in the application, the player may play past the duration you specified, which can be particularly noticeable when looping segments. Making the loop a high-priority callback will help, but for safety's sake, it's best to put a buffer of a few frames between segments.
  - If you're planning to switch between a series of movies in the same location on screen, don't move them to and from the space; instead, simply show or hide them.
  - If you have a choice between using bitmap-based animation or movie-player based animation, the movie-player option will likely be faster (because movement between frames is being controlled at a lower level, rather than through callbacks at the ScriptX layer), and take up less space (because of Cinepak compression).

  Problems to be aware of with the movie-player option:

- Invisible colors combined with Cinepak compression tend to leave edge noise. If your color scheme can't be adapted to the noise, you may have to fall back to bitmap animation.
- You have much less frame-by-frame control.

Disk Access

- If you need a group of objects, rather than loading them one at a time, load them all. If they are nested, use `loadDeep`.

Optimizing Memory

- Wherever possible, share objects instead of duplicating them.
- Constantly check “memory used” in Visual Memory to catch unnecessary memory use.
- Load and purge objects efficiently (load from disk, purge from memory)
- Windows consume a great deal of memory. Use only one window; prepend and delete spaces to that window.
- Delete and drop all references to saved objects (so they are freed) rather than hide or disable them to regain memory.

Memory Tools in ScriptX

- Visual Memory – Available by choosing **Visual Memory** from the File menu. It is also available in the Kaleida Media Player (see “Visual Memory” chapter for details).
- Functions to optimize memory. These are documented in the “Title Analysis API” chapter in this document.

  `allInstances` *className* – (core function) defined in `Behavior` class

  `disassemble` – (core function) not documented

  `findParents` *address maxLevel* – (core function)

  `treeSize` *object verbose maxDepth* – (core function)

  `objectSize` *maxDepth* – (core function)

  `memoryDiff()` – (scripted function) documented in the “Title Analysis API” chapter

  `memorySnap()` – (scripted function) documented in the “Title Analysis API” chapter

  `memoryUsage` *items* – (core function) documented in the “Title Analysis API” chapter

  `moo()` – (scripted function) documented in the “Title Analysis API” chapter

  `showCode` – (global variable)

Size of A Class

Classes take about 1K of memory each, not including method implementation. For this reason, when trying to conserve memory, it is useful to think about reducing the number of classes.

Garbage Collection of Persistent and Transient Objects

The following is a coherent way to view the memory system. All objects have two parts: a handle which points to its body. The handle is a kind of pointer to the body and is always 32 bits; the body is the rest of the object.

When you drop all references to the handle of any object, the handle and body are automatically garbage collected.

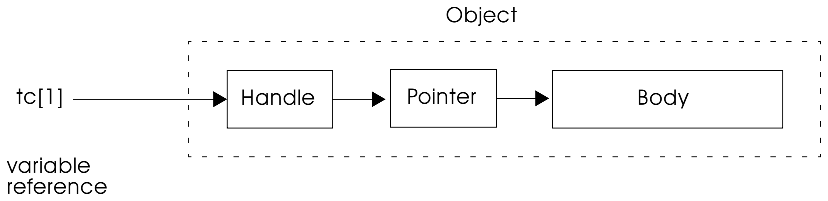


Figure 3-1: All objects have a handle and a body. The body can be freed when you call makePurgeable. The handle and body can both be freed only by dropped all references to both.

Essentially there are two separate worlds of objects: the "persistent" (stored) world and the "transient" (non-stored) world:

- A non-stored object is created by calling `new`. When there are no further references to it, it is automatically garbage collected. Because the object is not stored, the handle and body are always garbage collected together, as a pair.
- Stored objects are brought into memory from a storage container. A *handle* to a top-level object loaded from a container cannot be garbage collected, since the container holds onto it. (If you remove the object from the top level, then its handle can be garbage collected if there are still no other references to it.) A subobject can be garbage-collected if it is no longer reachable from core objects.

If you want to keep the handle of a stored object in memory (to maintain all its current references), but free its body, call `makePurgeable` on the object. Note this only marks the object, it will likely be purged by the end of the next complete garbage collection cycle. (When an object's handle is freed, it is garbage-collected at the next cycle; when an object is purged, its body is freed but not its handle.)

When you have a complete network of the persistent objects that can reach each other, and call `makePurgeable` on all those objects in the network, both the bodies and handles for all those objects will be garbage collected.

Any method call on a stored object after it has been marked as purgable will have one of two results (the one chosen is determined by gc latency and substrate reference optimization):

- If the object was actually removed from memory, it will be reloaded from the storage container before the method is called on it
- If the object has *not* been removed from memory, the purge bit will be cleared and the method will be called on the original object still in memory

You must be prepared to deal with either case. The first case can happen if the method call happened before the complete garbage collector cycle ran, or if in the substrate some code has a direct reference to the object's body.

To re-iterate, the purpose of `makePurgeable` is to maintain references to the object so that you can again pull it into memory (otherwise, you could just drop all references to it).

Rules of Thumb for Creating and Purging Objects

From this, we derive several rules of thumb for good title design:

1. If you have lots of small, short-lived objects, it can save small amounts of xmemory to instantiate them at runtime from a class rather than load them from storage, since their handles cannot be garbage collected once brought in from storage.
2. If you need to purge an object, it is safer to purge things are not changed, like bitmaps and audio, so the newly loaded version is identical to any memory-resident version.
3. If you purge an object that has changed state, you must be prepared to set its state to the desired values before you next use it, since it can be either in its most recently used state or its "fresh from store" state:
    - If you want to next use it in its "fresh from store" state, you can set its state to the stored state *before* you call `makePurgeable` on it.
    - If you want it to be in some other state, you must set its state *after* you touch it again but before you use it.

Scene Changes

Scene changes are inherently at odds with the design of the garbage collector. A scene change is any place in a title where in a short amount of time you want to remove a significant number or size of objects and then add other objects. Freeing the old objects can be done either of two ways, with the usual tradeoff between speed and memory:

- Free them all at once, or "synchronously", by calling `makePurgeable` (if stored) or dropping references (if non-stored) and then calling `garbageCollect`.

  This reduces the maximum memory requirement, but there will be a pause (about 1 second on a low-end machine) due to calling `garbageCollect`.

- Allow the garbage collector to free them when it gets to them, "asynchronously".

  This reduces the time to switch scenes, but requires more memory, as new objects are loaded before all the old ones are freed.

If you can afford the pause of an explicit call to `garbageCollect`, then the first approach allows all old objects to be freed before loading new ones. If that pause is too long, then you must try the second approach, allowing new objects to be loaded into memory before the old ones are all garbage collected.

Dropping Bitmap Data

If a bitmap has been loaded from a container such as a title container or library container, you can use the `dropData` method to force the bitmap to drop its data immediately.

When you make a bitmap purgeable, by calling `makePurgeable` on it, the garbage collector cleans up the memory that the bitmap uses. However, this does not happen instantly. It happens as soon as the garbage collector gets round to it. Before making a bitmap purgeable, you can call `dropData` on it, to immediately free up the memory used by the bitmap. When the garbage collector gets round to cleaning up the object, it completes the cleanup by freeing the memory occupied by the actual `Bitmap` object.

If you call `dropData` on a bitmap that has not been loaded from a container, (that is, you imported it in the same ScriptX session), you will get an exception.

Debugging Memory Problems

There are three main classes of memory problems in ScriptX applications:

- **Unexpected loading of objects** – You get loading of objects from storage containers at weird unexpected moments—for example, after you've already decided that you're done with the object and have purged it.

  Debugging approach: Specialize the `afterLoading` method on the object (with `print` statements, for example) to track inflations of the objects of interest to figure out what's triggering the inflations.

- **Unnecessary garbage produced in loops –** Unexpected amounts of memory is being eaten away when it doesn't seem like extra memory should be needed.

  Debugging approach: Use the `memorySnap` and `memoryDiff` functions to take a find out what objects have come into memory.

- **Objects don't get garbage-collected** – The garbage collector doesn't want to collect certain objects.

  Debugging approach: Use `fap` (find all parents) and `ppath` (parent path) functions to locate objects that are keeping otherwise-collectible objects around. Remember that browsers and the most recent expression in the Listener window will also keep objects in memory.

Ubiquitously useful are the familiar `allInstances` method and Visual Memory.

Loading and Purging

- Purging media – See the "Title Management" chapter in the *ScriptX Components Guide*.
- Make almost everything purgeable -- retrieval is unexpectedly fast
- Think in terms of scenes, collections and 1-of-n presenters

When purging, some caveats should be noted. In our Monterey example, there were several cases in which we load a `TwoDShape` object whose `presentedBy` flag was saved as `undefined`.

Then we append this shape to a space. Later, this shape is purged. However, the space referring to this shape can still refer to the shape. When this happens, the shape is reloaded. However, its `presentedBy` flag is reloaded as `undefined` when it should really be set to the space. This prevents the shape from ever getting updated.

Presenters and Bitmaps

- Use non-bitmap stencils (such as `Rect`) rather than bitmap stencils where equivalent.
- Trim bitmaps to the smallest possible bounding rectangle wherever possible.
- Use compressed DIBs with the low memory options whenever possible.
- Don't use full-screen bitmaps unless absolutely necessary. Use `Rect.`
- Use fills and small bitmaps as patterns whenever possible.
- Tile small bitmaps rather than using large bitmaps.

Minimizing The Need For Garbage Collection

When optimizing a title, it is important to minimize the amount of time the garbage collector needs to run. You can do this by following a few general rules, stated below, and then checking your title while it's running to make sure it's garbage collecting only where you know it needs to.

An object in memory becomes available for garbage collection when you are done using it (that is, you have dropped all references to it, or have called `makePurgeable` on it). Thus, an object that is no longer used becomes garbage. While it's a good idea in general to avoid creating unnecessary garbage, it can be more critical inside a loop. If garbage is created each cycle of a loop, the garbage collector will continue to be called while that loop is running, which steals processing time from other activities, such as compositing.

For example, some simple animations can be implemented by using a periodic callback that repeats an action on a regular basis. Periodic callbacks get called frequently, often many times a second. If each callback invocation generates just a tiny bit of unnecessary garbage, and is running 30 times a second, the garbage accumulates, and the garbage collector needs to run often to keep up with it. Since the garbage collector can use as much as 20% to 30% of the processing time, your animation could be running 20% to 30% less efficiently than it should.

So the basic aim is to minimize garbage generation within loops.

Exceptions can Prevent Garbage Collection

Each time an exception occurs, the `throwTag` and `throwArg` global variables get bound to objects. The `throwTag` is bound to the `Exception` object that was created, while `throwArg` is bound to the arguments passed to the `report` method (which reports the exception.) If an object, such as a window, is passed to an exception, then `throwArg` points to that window, and the window cannot be garbage collected.

To free up whatever objects are currently bound to `throwArg`, you can either set `throwArg` to undefined, or do something that causes an exception, such as `divideByZero`, where the `throwArg` does not hold onto an object that needs to be garbage collected.

For example, to cause a `divideByZero` exception, simply divide any number by 0, for example:

```
10 / 0
```

To Minimize Creating Garbage in Loops:

- Within a loop, avoid creating a new object that you will just free later in that loop. Re-use existing objects, and store them in instance variables if necessary.

  For example, if you think you need a new array each time a callback runs, instead create only one array, put it in an instance variable, then empty it out and fill it each time the callback runs. An array is a contiguous block of memory whose elements point to objects; as long as its number of elements does not change, the amount of memory it uses does not change. Emptying it out simply nulls out elements, rather than reduces its size.
- Within a loop, avoid defining an anonymous function which is not held onto by any variable. Because it has no reference, it gets garbage collected once each loop. Instead, predefine the function. For example:

  ```
  Avoid:  forEach myArray (value arg -> print value) arg

  Prefer: myFunc val arg -> (print val) -- Do this outside the loop
          forEach myArray myFunc arg    -- Do this inside the loop
  ```
- Within a loop, avoid using the “`for a in b do c`” syntax, since it creates and frees an iterator each time it is called. Instead, use the `forEach` method (which does not create an iterator) and give it a function to call on each member of the collection. Give it a named function, not an anonymous function, for the previous reason. <u>(Don't give it a method in or about build 40, or your title will crash.)</u>

```
Avoid:  for val in myArray do myFunc

Prefer: forEach myArray myFunc arg
```

Calling makePurgeable on an Object

The following rules are true inside or outside loops.

- Make sure you have completely finished using an object before you call `makePurgable` on it.

  For example, don't call `makePurgeable` on a window and then empty it out; do it the other way around. Using the object simply pulls it back into memory, defeating the effort to purge it. In addition, bugs might occur if the object pulled in from storage is in a different state than it was in memory (that is, if some of its transient instance variables were changed).

Visual Memory Tips

For details, see the "Visual Memory" chapter of this document.

Garbage Collection Tips

Up to 20% of the processor time can be spent doing garbage collection.

Garbage collection should occur only when freeing objects—not when showing or hiding objects, or moving them across the screen.

ShowCode Tips

To find out if some code implicitly creates an object (behind your back), set the global variable `showcode` to `true` and call the code to see what it does. This shows the individual, somewhat cryptic, bytecode instructions that get called. From this you might be able to guess if an object is being created.

For example, the `for` statement creates an iterator

```
showcode := true
for i in #(1) do 0
```

This prints out a lot of stuff in the Listener, including the following (Line 6 shows that an iterator is created):

```
0:  PUSH-EXTERNAL-CONSTANT undefined
2:  PUSH-ONE
3:  CALL-EXTERNAL Substrate:makeArrayValues 1
6:  CALL-EXTERNAL Substrate:iterate 1
9:  PUSH-LOCAL 1  (iIterator)
11: CALL-EXTERNAL Substrate:next 1
14: JUMP-IF-FALSE 26
```

```
16: PUSH-LOCAL 1  (iIterator)
18: CALL-EXTERNAL-DISCARD Substrate:valueGetter 1
21: PUSH-ZERO
22: POP-INTO-LOCAL 0  (result113)
24: JUMP 9
26: PUSH-LOCAL 0  (result113)
28: RETURN
```

Other

- If you have instance-specific information, design so that you can store it in instances rather than in rarely-instantiated class definitions.
- Often you can cut down on the number of presenters, controllers, spaces, and so on, by replacing "generic" classes (like `PushButton`) with specialized classes designed specifically for your application.

Reducing File Size

If your delivery mechanism is CD-ROM, file size of code is not usually a problem—the media usually fills up the disc. However, when doing `fileIn`, set the keyword `debugInfo:false` to reduce the file size of code significantly. This prevents the source code from being saved along with each bytecode method.

ScriptX Title Analysis API

4

This chapter describes functions and variables that can help ScriptX programmers analyze various characteristics of their titles, including memory usage, compositing performance, and other factors.

Some of these functions are defined in ScriptX, and are marked "global function". others are implemented as scripts that need to be loaded into ScriptX to be defined, and are marked "scripted global function."

Note-Do not call any of these functions or variables from within a ScriptX title that needs to run in the Kaleida Media Player. Many of these functions and variables are not available in the Kaleida Media Player. All are unsupported, and are not guaranteed to work on all platforms or in future versions of ScriptX. They are provided to assist you in the process of ScriptX title development.

disassemble (global function)

`disassemble` *funcObj* ⇨ `Boolean`

funcObj	Scripted `Function` or `ByteCodeMethod` to disassemble

Disassembles the scripted function or byte ode method *funcObj* and prints its bytecode representation in the ScriptX Listener window. The information presented by this function is in the same form as that displayed by the ScriptX Debugger. Note that this function works only with scripted, not core, functions or methods. This method returns false if *funcObj* cannot be disassembled.

To disassemble a method, call `methodBinding` (defined in `Behavior`) on the class and method you want to disassemble. For example:

```
disassemble (methodBinding myClass myMethod)
```

findParents (global function)

`findParents` *address maxLevel* ⇨ *(none)*

address	Address of object to find the parents of
maxLevel	`Integer` indicating how far up the reference paths to go

This function is used for debugging, when you want to find who's holding onto an object. This is useful, for example, if you expect the object to already have been garbage collected. (Removing the last reference to an object allows it to be garbage collected.)

The `findParents` function prints to the Listener window all references to the object with the specified *address*, recursing *maxLevel* deep. The *maxLevel* indicates how far to recurse up the reference paths, parent-to-parent-to-parent, and so forth. It stops recursing at roots (C globals, stack, global variables) and also when it detects cyclic references.

You can specify the integer address either by using the address value displayed in the return value of an object, using the `addressOf` function on an object, or using the address printed when you click on a memory segment in Visual Memory.

For example, to find all the variables and objects that point to `theTitleContainer`, 2 levels deep:

```
findParents (addressOf theTitleContainer) 2
```

The `findParents` function prints as complete a description as possible of the parents and the object being queried. This can be useful just for getting information about the object, since it describes persistence, object ID, host storage container, static/dynamic/system heap residience, if there's a t -> p present (internal debugging info), and so forth.

fps (global function)

`fps()` ⇨ `Float`

The "frames-per-second" function prints a float representing the current frame rate. This value is based on the number of times per second the compositor for the current window is compositing.

memoryDiff (scripted global function)

`memoryDiff` *memorySnap* ⇨ *(none)*

memorySnap `RamStream` object returns from `memorySnap` function

This function prints to the Listener window a list of the different objects between *memorySnap* and the current state of memory. Use this function after calling the `memorySnap` function. For example,:

```
global m := memorySnap()
-- Make some changes to memory, such as change scenes in a title
memoryDiff m
⇨ There are 4 more instances of Rect. There is a total of 14 instances.
```

The two functions `memorySnap` and `memoryDiff` work best when paired togehter. However, you can take several memory snaps, and then call `memoryDiff` on those snaps at different times.

This function is scripted. To define `memoryDiff`, run the script `memdiff.sx` located in the `utils/memory` folder on the ScriptX release.

memorySnap (scripted global function)

`memorySnap()` ⇨ `RamStream?`

This function uses `memoryUsage` to capture the topmost objects in memory. To take a snapshot of memory:

```
global m := memorySnap()
```

Then call `memoryDiff` to analyze this snapshot.

This function is scripted. To define `memorySnap`, run the script `memdiff.sx` located in the `utils/memory` folder on the ScriptX release.

memoryUsage (global function)

`memoryUsage` *num* ⇨ `Integer`

num `Integer` object

Reports how memory in the ScriptX heap is being used., based on the value of *num*:

- If *num* equals 0, then `memoryUsage` reports a simple summary.
- If *num* is a positive integer, it reports an ordered list of the largest *num* consumers of memory.

- If *items* is a negative integer, memoryUsage reports on consumers of static memory instead of dynamic (heap) memory.

Returns *num*, even if fewer actual consumers of memory were found. This function is for testing and debugging only, and should not be used in any title or library. *(Tools component)*

```
-- Example. Report top 30 consumers of memory in the ScriptX heap.
memoryUsage 30
-- prints report to window with up to 30 items
⇨ 30 -- returns 30
```

moo (scripted global function)

moo() ⇨ *(none)*

The “memory overhead overseer” function. This function is scripted; to define it, run the script listed at the end of this description.

This function is useful for finding memory leaks in a title. For example, type moo() after each scene change to see how much memory is being used. This function also keeps track of cumulative memory loss since you first ran moo.

To find out what memory is being held onto, use findParents global function and visual memory.

Here's sample output from moo:

```
moo()
"free sys "1507040"  heap used "1556720"  total heap "2709504
"delta free sys "-52736"  delta heap used "1200"  delta total heap "0
"cumul free sys "-29280"  cumul used heap "6112"  cumul total heap "0
"adj cumul free sys "-29280
```

The first line indicates the current figures.
The second line is the difference between current and previous figures.
The third line is the difference between current and starting figures.
The last line is free system memory adjusted for growth of the ScriptX heap.

This function is scripted. To define moo, run the script moo.sx located in the utils/memory folder on the ScriptX release.

The moo function uses the functions totalFreeSystemSpace, totalFreeHeapSpace and totalHeapSpace, which are documented in the “Global Functions” chapter of the *ScriptX Class Reference*.

ptt (global function)

ptt *filename* ⇨ *(none)*

filename	String representing name of file

Print thread timing. This function prints timing information in the ScriptX Listener and—in conjunction with the rtt function—to the *filename* file in theStartDir directory. For example:

```
createFile theStartDir "pttfile" @text
ptt "pttfile"
```

Information printed in the Listener window is formatted as follows:

```
   millisec       slices  time/slice  slice/sec
    4327  100.0%    343     12.6      79.3    Total
      57    1.3%      0      0.0       0.0   scheduler
    2463   56.9%     81     30.4      18.7   Idle
      34    0.8%      4      8.7       0.9   @theMainThread
       0    0.0%      0      0.0       0.0   "User Priority CallBacks"
      52    1.2%     18.9     0.3      43.7   "System Priority CallBacks""
       0    0.0%      0      0.0       0.0   "EventDispatchQueue"
    1448   33.5%     49     29.6      11.3   "Garbage Collector"
     268    6.2%    101      2.7      23.3   "ConsoleThread"
```

This report represents a snapshot of current thread timing information. The column “millisec” represents the amount of time consumed by each thread, and the percentage column (unlabeled) represents the percentage of total time taken per thread. “slices” represents the number of time slices allotted to each thread, “time/slice” represents the number of milliseconds per slice, and “slice/sec” represents the number of slices for a thread each second. The last column prints the names of each thread currently running.

Information printed in the *filename* file is formatted as follows:

```
2884321    2884321    0          "Idle"
2884505    184        0          "Scheduler"
-          -          enter      "System Priority CallBacks"
2884544    39         0          "System Priority CallBacks"
2884636    92         0          "Scheduler"
-          -          enter      "System Priority CallBacks"
2884855    219        0          "ConsoleCB"
2885142    287        0          "System Priority CallBacks"
2885354    212        0          "Scheduler"
-          -          enter      "ConsoleThread"
```

This information may not be as immediately useful as that printed to the Listener. The first column represents time in milliseconds at which a thread begins execution, the second represents the total time in milliseconds taken by a particular execution.

rtt (global function)

`rtt` *arg* ⇨ *(none)*

arg `Integer` value of `1` or `0`

Reset thread timing. Use this function with `ptt` to control the output of periodic thread timing information. An *arg* of `1` resets the time and causes the thread timing information to begin writing out to the file specified with `ptt`. An argument of `0` resets the time and causes thread timing information to stop writing out to the file specified with `ptt`.

showCode (global variable)

showCode ⇨ `Boolean`

When this variable is set `true`, and whenever expressions execute in ScriptX, the corresponding bytecode instructions and all intermediate forms of the parse tree display in the Listener window. This makes it possible to analyze ScriptX expressions. To turn it on, set it true:

```
showcode := true
```

PART TWO

Kaleida Media Player

CHAPTER

5

Kaleida Media Player User Guide

Kaleida Media Player User Guide 5

This chapter provides a sample "User Guide" that ScriptX title developers can modify for inclusion with their ScriptX titles. The audience for this User Guide is any end-users the need to install the Kaleida Media Player (KMP) and a title. This chapter is a template for telling your customers how to accomplish this installation, launch their title, and use the KMP menus.

NOTE – This chapter talks about an installer script for the Kaleida Media Player. Kaleida does not provide such an installer; you need to provide your own. When you set up your installer, be sure to include everything in your title's directory (KMPDIR) except for SX_Temp (which may include a symbol file, DLL files, and a file for a KMP icon), and of course include anything else your title requires such as loadables and importers.

This chapter uses the title *Your Example Title* as the example ScriptX title that your customer wants to install. Similarly, generic names in italics are used for names of installer scripts and files that your customers need to install.

NOTE – Feel free to duplicate all or any part of this chapter for your own use, as you see fit, substituting your own title's name in place of *Your Example Title* and your own installer name and other file names.

This documentation assumes that your title has implemented all of the features in the standard way, as documented. For example, your title must implement cut, copy, and paste to move a selection to and from the clipboard.

The remainder of this chapter contains the sample installation documentation, which covers four sections:

- Introduction
- Installation Instructions
- Starting the Kaleida Media Player
- Kaleida Media Player Menus

Introduction

The enclosed copy of *Your Example Title* includes the Kaleida Media Player (KMP), a software-only, cross-platform multimedia platform (no special hardware is required). Written in ScriptX, the programming language developed by Kaleida Labs, Inc., *Your Example Title* is one of the first of the new generation of applications being developed to play on the KMP.

Before you can play *Your Example Title*, the KMP must be installed on the primary hard disk drive of your multimedia personal computer. Following is a set of installation instructions, an explanation of how to launch *Your Example Title* on your Macintosh, Microsoft Windows, or OS/2 multimedia personal computer, and a quick reference to the menu items available to you from within the KMP.

Installation Instructions

To install the Kaleida Media Player (KMP) on your Macintosh, Windows, or OS/2 multimedia personal computer, follow the instructions given in this section.

Macintosh

1. Insert the *Your Example Title* CD-ROM into the CD-ROM drive. A CD-ROM icon entitled *Your Example Title* appears on your desktop.
2. Double-click on the *Your Example Title* icon. This opens a window with the contents of the *Your Example Title* CD-ROM displayed.
3. Double-click on the folder titled *mac.KMP*. This opens a new window that contains an icon named *Installr*.
4. Double-click on the *Installr* icon to install the KMP.

Windows

1. Insert the *Your Example Title* CD-ROM into the CD-ROM drive.
2. Launch the installer from either the Program Manager or the File Manager.

- To launch the installer from the Program Manager, select **Run** from the Program Manager File menu. At the dialog command line, type the letter of your CD-ROM drive, a colon, a backslash, *win.KMP*, a backslash, and `SETUP` (assuming the installer is named `SETUP.EXE`). For example , if your CD-ROM drive is drive D, type:

  ```
  D:\win.KMP\SETUP
  ```

  followed by the Enter key.
- To launch the installer from the File Manager, select the CD-ROM drive and the *win.KMP* directory. In the *win.KMP* directory you will find an icon for `SETUP.EXE`. Double-click on this icon to start the installer.

3. When the installer is launched, a dialog box appears. Click on the "*Set Location*" button.
4. Type the drive letter onto which you would like the KMP installed followed by *KMPDIR*, which is the name of the directory where you want the KMP installed. For example, if the hard drive where you would like to install the KMP is drive C, type:

   ```
   C:\KMPDIR
   ```

followed by the Enter key.

Once you have entered the name, click on OK. This returns you to the opening dialog box.

5. Click on the "*Install*" button.

OS/2

1. Insert the *Your Example Title* CD-ROM into the CD-ROM drive.
2. Launch the installer from either an OS/2 session or the File Manager.

- To launch the installer from an OS/2 session, either windowed or full screen, type the letter of your CD-ROM drive, a colon, a backslash, *os2.KMP*, a backslash, and `SETUP` (assuming the installer is named `SETUP.EXE`). For example , if your CD-ROM drive is drive D, type:

```
D:\os2.KMP\SETUP
```

followed by the Enter key.

- To launch the installer from the File Manager:
  a. Double-click on your desktop Drives folder.
  b. Double-click on the icon for your CD-ROM.
  c. Double-click on the icon for your *os2.KMP* directory.
  d. Double-click on the icon for `SETUP.EXE`.

3. When the installer is launched, follow the on-screen instructions.
4. *** DLL ??? ***

Starting the Title

The title file appears as an icon in the operating system as follows:

(Your icon goes here)

The KMP file appears as an icon in the operating system as follows:

Macintosh

You can start your title on the Macintosh in any one of the following ways:; these options open both the title and the KMP:

- By double-clicking the icon for your title file, *Your Example Title*
- By dragging the icon for the title file onto the KMP icon

Windows

You can start the title on Windows from either the File Manager or the Program Manager; these options open both the title and the KMP:

- From the Program Manager, double-click on the *Your Example Title* icon.
- From the File Manager, double-click on the *ExmplTtl.sxt* file within the appropriate directory.

OS/2

You can start the KMP on OS/2 from either an OS/2 session or the File Manager; these options open both the title and the KMP:

- From the File Manager, double-click on the *Your Example Title* icon.
- From within an OS/2 session, double-click on the *ExmplTtl.sxt* file within the appropriate directory.

Kaleida Media Player Menus

This section describes each of the commands available from the menus of the Kaleida Media Player (KMP).

The File Menu

The File menu provides access to KMP titles, as well as a way to close title files and quit the Kaleida Media Player application.

Open Title

Opens KMP title files such as *Your Example Title*.

A title file is a file that contains a complete, working title developed for the KMP. When you open a title file, the title starts and runs. Note that you can also open a title file by double-clicking its icon or by dragging its icon onto the KMP icon (Macintosh only). Filenames for titles generally end in "`.SXT`".

Open Accessory

Opens KMP accessory files.

An accessory file is a file that contains a KMP accessory. Accessories are not intended to run on their own. Instead, they are designed to be used within other titles. An accessory closes when you close its title, unless another title is also using it. Filenames for accessories generally end in "`.SXA`".

Close Title

Closes the current title.

Page Setup or Print Setup

Displays the Page Setup (Macintosh) or Print Setup (Windows) dialog box that allows you to set options for printing.

Print

Provides standard printing features for KMP titles.

Quit or Exit

Quits the Kaleida Media Player, closing all titles and accessories previously opened. This menu command is named **Quit** on the Macintosh and **Exit** in Microsoft Windows and OS/2.

About ScriptX

Provides version and copyright information for the KMP. On the Macintosh, this menu item is located as the first item under the Apple menu.

The Edit Menu

Provides standard editing features for text in KMP titles.

Undo

(Not yet implemented.)

Cut

Cuts the current selection, which removes it from its current location and places it in the clipboard.

Copy

Copies the current selection, which leaves it undisturbed and places a copy of it on the clipboard.

Paste

Pastes the current content of the clipboard at the current insertion point or selection.

Clear

Clears the current selection, removing it from its current location and discarding it. Has no effect on the clipboard.

The Window Menu

This menu provides a list of all windows currently open in the KMP, listing them by name. This menu is empty until you open a title with a window. Selecting a window from this menu brings it in front of all other windows on the screen.

In Microsoft Windows and in OS/2, this menu contains three extra commands: **Cascade**, **Tile**, and **Arrange Icons**. These are standard menu items for those systems and are described in the documentation for those systems.

PART THREE

Tools

CHAPTER

6

Introduction to Tools

This part of the *ScriptX Developer's Guide* discusses the ScriptX development environment and the windows, browsers and tools available to help you use the system and view your objects.

This part includes the following chapters:

- Chapter 7, “ScriptX Listener and Menus,” introduces the ScriptX Language and Class Library and describes how to use the ScriptX Listener window to evaluate and load ScriptX code.
- Chapter 8, “The Browser” discusses the browsers that provide graphical interfaces for viewing classes, instances, functions, free methods, and modules.
- Chapter 9, “Debugger,” discusses the debugger which you can use to investigate exceptions and step through code.
- Chapter 10, “ByteCodeMethod Profiler” discusses how to use the ByteCodeMethod profiler to see where your methods are spending most of their time.
- Chapter 11, “Visual Memory” describes how to use the Visual Memory Tool.
- Chapter 12, “Tool Framework describes the platform on top of which tools can be built.
- Chapter 13, “Tool Framework API” describes the classes, methods, functions and variables that support the tool framework of the previous chapter.
- Chapter 14, “Photoshop Plug-ins for KIC Compression” describes how to use the Kalieda Image Compression (KIC) plug in for Photoshop.
- Chapter 15, “Importing Media,” describes how to use import non-ScriptX media files into ScriptX. The imported media is converted into appropriate ScriptX objects. For example, QuickTime™ movies can be imported as `MoviePlayer` objects; AIFF sound files can be imported as `DigitalAudioPlayer` objects; and image files can be imported as `Bitmap` objects.
- Chapter 16, “Using the Director-to- ScorePlayer Importer,” describes how to import titles built with Macromedia Director® into ScriptX, and how to modify these imported titles in ScriptX.
- Chapter 17, “Director-to-ScorePlayer Importer API,” describes the API used by the classes that import and play Director titles in ScriptX.
- Chapter 18, “Using The Director Translation Kit,” describes how to build your own customized importer for importing Director titles.
- Chapter 19, “Director Translation Kit API,” describes the API used by the classes that enable you to build customized Director importers.

Loading Tools and Importers

This section discusses how to select which tools and importers to load when ScriptX starts up. In the current release, you cannot load a tool or importer after ScriptX has started.

Loading Tools

To determine what tools are loaded when ScriptX starts up, move tools in or out of the `Tools` directory in the ScriptX startup directory.

When ScriptX starts up, it looks for a directory called `Tools` in the ScriptX startup directory. If this directory contains a library container called `toolutil.sxl` and `widgets.sxl`, ScriptX loads the tools in the `Tools` directory. It then loads the tools in each first-level subdirectory of `Tools`, processing the directories in alphabetical order.

If a subdirectory in the `Tools` directory contains its own subdirectories, the contents of those subdirectories do not get loaded automatically. You must move all tools that you want loaded to the `Tools` directory or to a first-level subdirectory of `Tools`.

If you don’t want all the tools loaded, you should move all the tools that you don’t want loaded out of the `Tools` directory and its first-level subdirectories. One way to move tools so they are not loaded, but are still located conveniently, is to create a directory containing a subdirectory in the `Tools` directory, and move all the tools that you don’t want loaded into the second-level subdirectory.

If ScriptX loads any tools, an additional `Tools` menu appears in the ScriptX menu bar. The commands available through this menu depend on which tools are loaded. The menu has a command for starting or opening each tool that has been loaded. When a tool is invoked, it may cause additional tool-specific menus to appear.

Each tool that is loaded adds memory overhead to the ScriptX application, so you may find you need to increase the memory for the ScriptX application as you increase the number of tools loaded.

After ScriptX has started, you can load individual tools by using the **Open** menu command to open the appropriate library container that loads tools.

You cannot load or use tools in the Kaleida Media Player.

Loading Importers

If you want to load importers, the ScriptX startup directory must contain a directory called `importrs` containing the importers to be loaded.

ScriptX is shipped to you with an `importrs` directory in the startup directory. This directory contains all the importers provided with ScriptX. If you do not want the importers to be loaded when ScriptX starts up, either rename the `importrs` directory to something else, such as `importrx`, or move the importers that you don’t want loaded out of the `importrs` directory.

The `importrs` directory contains a subdirectory named for the platform you are using (for example, `mac` or `win`). Each platform subdirectory contains the loadable files that implement the import and export capabilities. (In the current release, there is only one exporter, which is a text exporter.)

When the ScriptX development environment starts up, it searches the `importrs` directory, and loads the importers and exporters that are appropriate to the platform and current software version.

The code for the importer and exporter routines is not loaded; instead, the kinds of conversions supported by the importer and exporter modules are registered in the system. The import/export engine loads the code for a specific conversion only when it is needed, not before. Thus, loading importers when ScriptX starts up does not have a significant memory overhead.

The recommended process for using media in titles is to import the media in the ScriptX development environment and save it to a container. If a title needs to use media at playback time, it can get it out of the container. Thus a title would generally speaking not use the importers in the Kaleida Media Player, although it is entirely possible and acceptable to do so.

CHAPTER

ScriptX Listener and Menus

7

The ScriptX Language and Class Library is implemented for the development of titles and tools in an executable file named “ScriptX.” This implementation has all the capabilities of the Kaleida Media Player (KMP), plus the compiler and other capabilities that support title and tool development. These capabilities are provided through the user interface of the ScriptX Listener window and ScriptX menu items.

This chapter lists the features that distinguish ScriptX from the KMP, describes how to start ScriptX, and then describes how to use both the ScriptX Listener window and the menu commands.

Features in ScriptX That Are Not in KMP

ScriptX has the following features not found in the Kaleida Media Player. In general, the API for these features are documented in the *ScriptX Class Reference* (and are noted as not being available in the Kaleida Media Player):

- ScriptX includes the ScriptX bytecode compiler, the component that compiles ASCII text scripts into bytecode format. ScriptX is thus able to accept, compile, and execute scripts written using the published ScriptX API published in the *Field Guide to the ScriptX Language* and the *ScriptX Class Reference*. On the other hand, the KMP can execute only code that has previously been compiled, such as the code saved in a title container.
- In ScriptX, the `fileIn` method, defined by the `DirRep` and `ByteStream` classes, provides access to the ScriptX bytecode compiler. Since the compiler isn’t present in the KMP, you can only use the `fileIn` method in code intended to run in ScriptX.
- In ScriptX, the global variable `theScriptDir` represents the directory where the most recently loaded and compiled script file resides. Since the KMP can’t load and compile scripts, the runtime doesn’t define this global.
- ScriptX can automatically load media importers, browsers, and other loadable development features described in this document. If your title needs to import media, you should explicitly include the appropriate importers on your release media and load them into the KMP from your title.
- In ScriptX, `showChangedRegion` is a diagnostic instance variable in the `TwoDCompositor` class that displays the changed regions of a window. This enables you to optimize your title to do the least amount of updating possible.
- ScriptX contains many other diagnostics associated with optimizing and debugging your scripts. The `warning` function is a diagnostic that warns you of conditions that can slow down performance. Currently it warns you when a bitmap transfer occurs with mismatched color maps.

The Development Process

Figure 7-1 illustrates the development process from the initial script (with an extension `.sx`), being filed into the compiler on Macintosh or Windows. Though greatly simplified, this figure shows the basic steps involved.

When you develop a ScriptX title or tool, you develop scripts using the ScriptX Language and Class Library for playback on the Kaleida Media Player.

The compiler parses the script file and generates bytecode, which can be saved to a title container (with extension `.sxt`), library container (`.sxl`), or accessory container (`.sxa`). This bytecode is portable across all platforms and can be played on either the Macintosh or Windows version of the Kaleida Media Player. The development process is described in more detail in the "Title Design" chapter in the binder documentation.

Notice the bytecode is shared by both the authoring and playback processes.

This process becomes more complicated when there are multiple script files, library files, and accessory files.

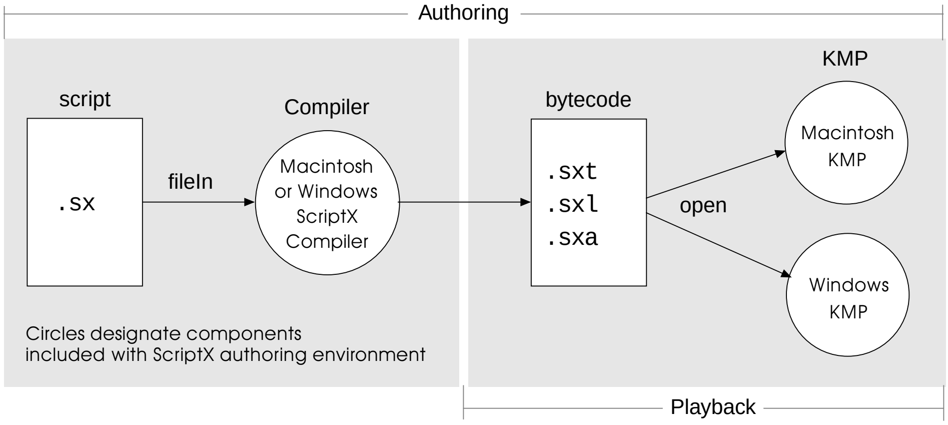
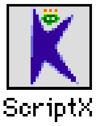

Authoring
script
.sx
fileIn
Compiler
Macintosh or Windows ScriptX Compiler
bytecode
.sxt
.sxl
.sxa
open
KMP
Macintosh KMP
Windows KMP
Circles designate components included with ScriptX authoring environment
Playback

Figure 7-1: The ScriptX authoring and playback processes

Starting ScriptX

The ScriptX executable file appears as an icon in the operating system as follows:



You can start ScriptX from the operating system in one of several ways:

- By double-clicking the ScriptX icon
- By clicking once on the ScriptX icon and choosing "Open" from the menu
- By dragging the icon for a title file onto the ScriptX icon, which also opens the title (Macintosh only)

Double-clicking on a title file will open the Kaleida Media Player, not ScriptX.

When you start ScriptX, you'll see the startup screen first, followed by the Listener window, labeled "ScriptX™ Listener," as shown in Figure 7-2. As ScriptX continues to load, you'll see some messages from importer and tool files, if any, as they are automatically loaded. The importers are implemented as ScriptX libraries that load non-ScriptX resources. The messages are of the form "SNDImporter registered," "DIBImporter registered," and so forth. Only importers listed in these messages are available for this session, unless they are a static part of the environment.

For a description of how to configure ScriptX to load various importers and tools, see "Loading Tools and Importers" on page 60.

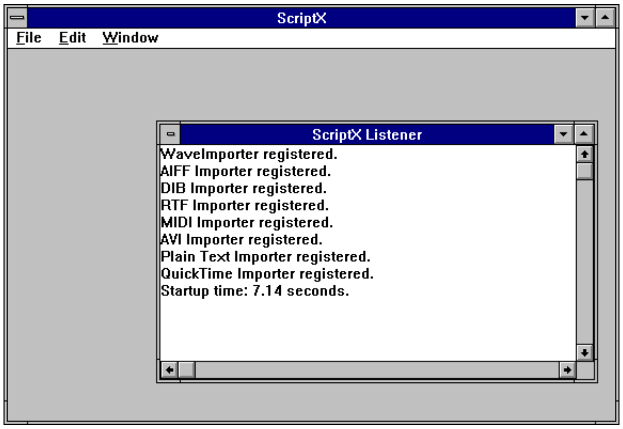
Note – If you want to quickly empty out the Listener window at any time (such as at startup), select all of the text in the window and then delete it:
Macintosh: Press Command-A and Delete
Windows: Press Ctrl-A and Backspace

The ScriptX Listener

If you've read the *ScriptX Quick Start Guide* and run the examples, you've already been introduced to the Listener. This discussion goes into greater detail of its workings.

As shown in Figure 7-2, the ScriptX Listener window is a text window in which you can interact with ScriptX. You can enter expressions into this window and ScriptX responds with results of the expressions.

The Listener itself is a process underlying the Listener window that accepts text expressions that you enter. It "listens," in a sense, for complete expressions that you enter, and responds by compiling and evaluating them. More specifically, the Listener is a continuously-running loop that parses the input stream, compiles it into bytecode, calls the bytecode to be evaluated, prints the returned result, then waits for the next input.



```
WaveImporter registered.
AIFF Importer registered.
DIB Importer registered.
RTF Importer registered.
MIDI Importer registered.
AVI Importer registered.
Plain Text Importer registered.
QuickTime Importer registered.
Startup time: 7.14 seconds.
```

Figure 7-2: ScriptX as it appears at startup with the Listener window open (Microsoft Windows version shown)

The Listener window can be thought of as the user interface to the ScriptX Language and Class Library. Through the Listener window, you gain direct control of the full facilities of ScriptX. You can use the Listener window to try out single expressions, and to test and debug full scripts. You can create objects and save them to title container files. Unless you're using a tool that has "Save As ScriptX" capability, you'll use the Listener extensively in developing a ScriptX title.

Developers have basically two ways to enter scripts for the Listener to evaluate:

- By typing or pasting text directly into the Listener window
- By typing text into a text file created in a separate text editor, such as TeachText on the Macintosh or NotePad on Windows. This text can be loaded either using the `fileIn` method or the **Open Title** menu command.

In either case, each complete ScriptX expression is compiled and executed immediately when the compiler recognizes the expression is complete. Each time a statement executes successfully, the Listener displays the return value. If a statement fails to execute, the Listener displays the exception that caused the failure.

Typing Scripts Directly into the Listener

The Listener Window is a simple text window. Any text that you type remains in the window. As you enter more and more text, the window eventually scrolls. You can scroll backward to find and re-execute previously entered scripts. You can execute a single line or select a group of lines and execute them all at once. The Listener lets you perform all of the standard text editing functions—you can select text and choose Cut or Copy from the Edit Menu to place it on the clipboard, and choose Paste to paste it elsewhere.

Note – Enter and Return are two different keys that do slightly different things in ScriptX. Windows computers label both keys “Enter.” This manual uses the following Macintosh convention (since it gives them different labels):
- The key located just above the right-hand Shift key is called “Return”
- The key located on the numeric keypad is called “Enter”

To enter scripts into the ScriptX Listener, just type the script and press Return after each line. ScriptX evaluates an expression when Return is pressed if the expression is complete. Thus

```
x := 1 <return>
```

returns the value 1 after you press `<return>`. In contrast,

```
x := <return>
1 <return>
```

returns the value 1 after you press the second `<return>`. It does not evaluate after the first `<return>` because an expression that ends with the assignment symbol (`:=`) is not complete. The Listener waits after the first `<return>` for you to complete the expression.

When pressed at the end of a line, the Enter key does the same thing as the Return key.

Throughout the rest of this manual, we do not show the `<return>` character in expressions. We assume that you will enter the line using either Return or Enter, or one the other techniques described later. When an expression returns a value, we use the arrow (⇨) in this manual as a convention to indicate the value returned. The previous expression returns the value 1 only after the 1 is entered, so would appear as follows:

```
x :=
1
⇨  1
```

The Continuation Character

When a ScriptX statement requires more than a single line, and if you simply press Return at the end of the first line, the Listener will determine whether the expression is minimally complete, and, if so, will compile and evaluate it. (If the expression is not complete, the Listener will wait for you to enter another line.) To avoid this, type the backslash continuation character, \, at the end of a line to signal to the Listener that it shouldn’t attempt to evaluate the current line yet. The following example illustrates:

- Incorrect:

```
global myShape := new TwoDShape target:(new Rect x2:100 y2:100)
fill:blackBrush
```

- Correct:

  ```
  global myShape := new TwoDShape target:(new Rect x2:100 y2:100) \
  fill:blackBrush
  ```

The first line of the incorrect version creates `myShape` with `fill` set to `undefined`, its default. The second line reports an error:

```
global myShape := new TwoDShape target:(new Rect x2:100 y2:100)
⇨  TwoDShape@0x14f2b08
fill:blackBrush
⇨  -- ** Syntax error: ("syntax error") at "fill:" (SyntaxError)
```

Incomplete Expressions

The Listener does not have any particular "command line" like some compliers do. You can type or select an expression on any line and press Enter to execute it. Control alternates between you and the Listener—you enter an expression, it evaluates the expression and returns control back to you.

The way you tell whether the Listener has returned control to you or not is by whether it has returned a value. If it has no value to return, it prints "OK." Sometimes you may find you enter a script that unexpectedly hangs up and does not return a value. In general, this is due to the expression not having been completed (possibly a bug in your script). The Listener is merely waiting for you to type the characters that will complete the expression.

- There is no guaranteed way to force the Listener to return control to you (such as if you have a runaway thread); however, if you are stuck in the middle of an expression, the first thing to try is typing two exclamation marks and press Enter, which forces the expression to terminate:
  ```
  !!
  ```

The following conditions can cause the Listener to "hang":

- An unbalanced pair is missing its trailing character:
  ```
  ( [ {
  ```
- A binary operator has been entered and is waiting for a value:
  ```
  1 +
  ```
- A definition or case statement has not been ended:
  ```
  case of
      (a = true): print "true"
      (a = false): print "false"
  ```
- A comment has been inserted in the wrong place. Inspect all comments

Executing A Previously Entered Line

You can re-execute an expression you have already entered in the Listener window. Do this by scrolling back to that line with the mouse or cursor keys, put the insertion point anywhere on the line, and press the Enter key (on the numeric keypad). This key always interprets the current selection.

Alternatively, you can move the insertion point to the end of the line and press Return to evaluate it (the same as if you had typed in the line). However, pressing a Return anywhere else but at the end of a line inserts a carriage return and does not attempt to evaluate the line.

Executing More Than One Line

To execute a group of lines that already appear in the Listener window, select them with the mouse and press the Enter key (on the numeric keypad). All lines are evaluated, and all results from the expressions appear at the end of the selected text.

Use this technique when you paste scripts into the window. For example, after you copy a function from a text editor and paste it into the Listener window , select the lines with the mouse, and press Enter to evaluate them.

Summary of Return and Enter Keys

The Return and Enter keys work as follows:

Return key	Evaluates a line only if the insertion point is at the end of the line; otherwise, it inserts a carriage return
Enter key	Evaluates a line when the insertion point is anywhere on the line; never inserts a carriage return

To evaluate an expression, do one of the following:

- For a single line with the insertion point anywhere, press the Enter key
- For a single line with the insertion point at the end, press the Return key
- For more than one line, select the lines and press the Enter key

If the line contains a complete expression, it is immediately evaluated; otherwise, the Listener continues to wait for you to enter the rest of the expression.

Creating and Compiling Script Files

While you can type simple scripts directly into the ScriptX Listener, to write a more serious script—one that you can save, modify, enhance, and debug—you create a script file. A script file is any standard ASCII text file created using a text editor such as TeachText on the Macintosh or NotePad on Windows. ASCII text is plain text without font styles applied to it, and is sometimes known as "Text only."

The procedure for compiling such a script file is to either load it from the menus with **Open Title** or from the Listener window with the `filein` method.

Note – The convention for naming script files, valid across all platforms, is to use the `.SX` extension. This extension is required in Microsoft Windows and optional on the Macintosh.

Loading Script Files Using 'fileIn'

You can use `fileIn` (a `DirRep` method) in the Listener window to open a script file. To load the file `sample.sx`, from the directory where ScriptX is located, do the following:

```
fileIn theStartDir name:"sample.sx"
```

This method opens the file, then reads in and evaluates each line, continuing non-stop until the end of the file. It displays only the value from the last evaluation.

The `fileIn` method is used in “make” files or “build” files, which are files that automate the building of titles made from multiple script files. A make file loads each of the script files in the proper order. Loading a make file is a way to do batch processing of scripts.

The following example is a make file that uses the `fileIn` method to load in several media files:

```
fileIn theStartDir name:"bitmaps.sx"
fileIn theStartDir name:"debug.sx"
fileIn theStartDir name:"panel.sx"
fileIn theStartDir name:"nodesObj.sx"
fileIn theStartDir name:"activate.sx"
```

A make file can either be loaded using the **Open Title** menu command or by again using `filein` on the name of the make file.

Note – The `fileIn` method is used only during the development of titles and is not available in the Kaleida Media Player, since this method uses the ScriptX compiler.

Loading Script Files From the Menu

The easiest way to open a script file created outside ScriptX is by selecting the **Open Title** command from the File menu. This menu command opens up a dialog box from which you can select the script and open it. When you open the file, it is loaded into the Listener window a line at a time by default—you must press Return to evaluate the displayed line and continue to the next line. This is called single-step compiling. You can change this to load the entire file without stopping, called quick-compiling.

In either case, when the end of the file is reached, the Listener window displays an end of file marker to indicate it is done:

```
<eof>
```

Quick-Compile and Single-Step Compile

You switch between these two compile modes by typing special symbols at the point in the script where you want to start quick-compiling or single-step compiling. Once you use a special symbol, that mode becomes the default. The rest of that file and any subsequent script files will read in with that mode until a symbol is encountered that changes the mode.

The two ways to compile scripts from the **Open Title** menu command are as follows:

- Quick-compile – Loads the entire file without pausing, like the `filein` method. The quick-compile symbol is:

  ```
  -- <<<
  ```

- Single-step compile – Loads a line at a time, which allows you to watch the value returned from each complete expression. This is the default mode. The single-step compile symbol is:

  ```
  -- >>>
  ```

In the single-step mode, the Listener reads in and displays the first line of your script without evaluating it yet, then pauses, waiting for you to press Return. This allows you to read each line before it runs. Once you press the Return key, it evaluates that line, reads in the next line and pauses again.

Notice these symbols are preceded by the comment symbol (`--`). The actual quick-compile and single-step symbols are <<< and >>>, respectively, made from three "less-than" or "greater than" signs. These two symbols are not a part of the ScriptX language, and are interpreted only by the **Open Title** menu command, not by the `filein` method. If you use the `fileIn` method to open such a file, you must put the symbols inside a comment, or it will report an error. The **Open Title** menu command reads the quick-compile symbol anywhere it occurs, even inside a comment. (If you will not use `fileIn` on that file, you can omit the comment symbol.)

Note – The **Open Title** menu command serves two distinct purposes: to open and compile scripts, as described here, and to open title files that are compiled and saved in title containers, as described in the Title Management chapter of the *ScriptX Components Guide*.

Using the Compile Symbols

If you want to troubleshoot just one particular part of a script, you can speed up the compile process: Put a quick-compile symbol at the beginning of a script, a single-step symbol just ahead of where you want to watch the expressions, and a quick-compile symbol after that section, as follows:

```
-- <<<

--     (This code will compile quickly)

-- >>>

--     (This code will be compiled one line at a time)

-- <<<

--     (This code will compile quickly)
```

Note – The single-step and quick-compile symbols are specific to the ScriptX Listener and are not part of the ScriptX Compiler. Thus, scripts loaded in other ways, such as through the `fileIn` method, will generate compiler errors if the characters are not embedded in a comment.

Quick Compiling With Intermediate Results

When the Listener encounters the symbol <<<- in a script, it begins compiling the script quickly, displaying the intermediate results. Note the hyphen (`-`) typed after the normal quick-compile symbol. In this mode, the Listener lists the return values and exceptions as each statement compiles, as well as any change in quick or single-step compiling. The complete symbol, including the comment symbol, is:

```
-- <<<-
```

Trying out the Sample Scripts

The ScriptX Developer's CD contains a number of well-commented script examples that you can run in the Listener window to learn more about ScriptX. You can also modify the scripts in these examples to fit the needs of your titles.

New Listener Windows

You can open other Listener windows by choosing **New Listener** from the File menu. This would be useful if you have two different scripts that you are working on simultaneously. Each Listener runs in a separate thread, so you can be in the middle of one expression in one window, and switch to the other to enter a separate expression. In addition, each Listener window can be switched to a different module. If a title has multiple modules that you are working in, it is convenient to open a new Listener window for each module and use the `in module` construct to switch modules.

You hide and show the original Listener window without losing its contents by choosing **Listener** from the File menu. New Listener windows cannot be hidden and shown—if you close one, it cannot be re-opened. New Listener windows can be closed by clicking in the close box (Macintosh) or choosing Close from its menu (Macintosh). This closes the window and purges it for garbage collection. Choosing **New Listener** opens a new Listener window.

Resetting ScriptX

As you execute ScriptX statements or run scripts from the ScriptX Listener, you alter the state of ScriptX. If you want to restore ScriptX to its original state, select the **Quit** or **Exit** command from the File menu, then restart it.

The Listener and Other ScriptX Windows

Any window, top clock, or top player that you create must belong to a title container. The global constant `theScratchTitle` is the default owner if you don't specify another. However, the Listener window is an exception to this rule. It doesn't belong to any title container, not even the scratch title, nor does it show up in any `TitleContainer`'s list of windows.

While you can always get to the Listener window from the Window menu, there are certain restrictions on interaction between the Listener and any titles you may be working on. These are discussed in the following sections.

Copying and Pasting Text in the Listener

You can use the **Cut**, **Copy**, and **Paste** menu commands in ScriptX to cut, copy and paste text from the Listener to and from other ScriptX applications. You would only be able to use these edit features with a ScriptX title if that title had implemented the `cut`, `copy`, or `paste` methods you want to use. For example, to cut from the Listener window and paste into a title window, that window would need to have the `paste` method implemented.

If you want to put `@native` text on the clipboard, you cannot do it from the Listener. ScriptX considers text copied or cut to the clipboard from the Listener window to be of type `@text`, which is the same as if it were copied or cut from a non-ScriptX application.

The Listener and User Focus

The presence of the Listener window does not disturb the user focus attributes of other windows or titles. That is, the value of the hasUserFocus instance variable of a title container and window does not change when the Listener is opened, brought in front, typed in, or closed, even though the title's window appears to have lost user focus behind the Listener window.

Exceptions in the Listener Window

In general, when an exception occurs in the Kaleida Media Player, an exception dialog box should appear. However, in this release only, exceptions that would appear in the Listener window in ScriptX do *not* display an exception dialog box. This means that if an exception can stop the KMP without displaying an exception dialog box. The KMP will simply hang and not continue.

To work around this, check carefully for exceptions that are reported to the Listener window, and work to prevent those exceptions.

Working With Other Windows (Macintosh only)

Instances of NoticeWindow, DialogWindow and PaletteWindow on the Macintosh display in a layer in front of the Listener window, and nothing you can do will move the Listener window in front of them. The Listener window may also appear inactive (its title bar is dimmed). However, that does not need to stop you from continuing to type into and use the Listener window.

To use the Listener when it cannot be moved forward, press Command-L, the keyboard command that normally makes the Listener active, then begin typing. Even though the Listener remains behind the window, you can continue to execute commands, such as calling hide on the window.

Before displaying the notice, dialog, or palette window, it helps to position the Listener window so that the window will not obscure it.

For example, this script creates a notice window, which is modal:

```
global n := new NoticeWindow
show n
-- Press Command-L
hide n
```

ScriptX Menu Command Reference

This section describes each of the commands available in the ScriptX menus. Menu items available only in Microsoft Windows or Macintosh are marked as such.

Some of these menu commands have keyboard shortcuts. If the menu bar is hidden, it is quite convenient to access some those menu commands by using their shortcuts.

The File Menu

The File menu provides access to ScriptX titles and script files, as well as a way to close title files and quit ScriptX. The File menu also provides access to two windows, the ScriptX Listener and ScriptX Visual Memory, that help you use ScriptX.

Open Title

Performs one of two distinct actions, depending on the file the user selects: Opens a title file that is compiled and saved in a title container (`.SXT`), or opens and compile a script file (`.SX`).

A title file is a file containing a `TitleContainer` instance and the objects representing a title. When you open a title file, the title starts and runs, as it would in the Kaleida Media Player. See the chapter "Title Management" in the *ScriptX Architecture and Components Guide* for more on creating and saving a title. Note that you can also open a title file in the operating system by dragging its icon onto the ScriptX icon (Macintosh only). In Windows, a title file must have a `.SXT` extension or it cannot be opened; the Macintosh has no such restriction.

When the user selects a title file, this menu command calls the `open` class method on `TitleContainer`, supplying the `dir` and `path` keywords with the directory and filename the user selected:

```
open TitleContainer dir:myDirRep path:filename
```

A script file is an ASCII text file containing a script written in the ScriptX language. You create script files using a standard text editor, such as TeachText on the Macintosh or NotePad in Windows. A script file is loaded one line at a time. As each line is loaded, it is compiled and evaluated by ScriptX. The Listener window displays each line as it loads, along with the return value of that expression. If an expression fails to compile, the Listener reports the exception that caused the failure. For more information, see "Loading Script Files From the Menu" on page 71. (This part of the **Open** menu command is not available in the KMP.) In Windows, a script file must have an `.SX` extension or it cannot be opened; the Macintosh has no such restriction.

This menu command calls the `fileIn` instance method on the instance of `DirRep` corresponding to the directory and filename the user selected:

```
fileIn myDirRep name:filename
```

Open Accessory

Opens a ScriptX accessory file (`.SXA`).

An accessory file is a file containing an `AccessoryContainer` instance and the objects representing an accessory. When you open an accessory file, it is made available to the current title. An accessory closes when you close its title, unless another title is also using it. See the chapter "Title Management" in the *ScriptX Architecture and Components Guide* for more on creating and saving an accessory.

This menu command calls the `open` class method on `AccessoryContainer`, supplying the `dir` and `path` keywords with the directory and filename the user selected:

```
open AccessoryContainer dir:myDirRep path:filename
```

Close Title

Closes a ScriptX title. If the scratch title is the current title container, then this command instead closes its frontmost window (you cannot close the scratch title). This command has no effect on the Listener window.

This menu command calls the `close` instance method on the current title container:

```
close theTitleContainer
```

Listener

Shows or hides the Listener window. See the discussion "The ScriptX Listener" on page 67 for more information.

If the Listener window is showing but has other non-modal windows in front of it, this command brings it in front of those windows. If the Listener window is the frontmost window, this command hides it. If the Listener window is currently hidden, this command shows it. (Not available in the KMP.)

Note – Macintosh only – If a dialog, notice, or palette window is in front of the Listener window, this command gives the Listener window typing focus so you can type commands into it. Notice that this menu command does not move the Listener window in front of the other windows.

New Listener

Creates and displays a new Listener window. Unlike the original Listener window, you cannot hide and later show this window—closing this window makes it available for garbage collection. See the discussion "New Listener Windows" on page 73 for more information. (Not available in the KMP.)

Visual Memory

Shows or hides the Visual Memory window, which shows a map of available memory in various colors. The window has a legend at the top that indicates what the colors mean. See the chapter "Memory Management" in the *ScriptX Architecture and Components Guide* for more on the use of the Visual Memory window.

If the Visual Memory window is currently visible, this command hides it. If the window is currently hidden, this command shows it. (Not available in the KMP.)

Page Setup or Print Setup

Displays the Page Setup (Macintosh) or Print Setup (Windows) dialog box that allows you to set options for printing. This option is not yet implemented for OS/2.

This menu option does not invoke any ScriptX methods. This menu option is a companion to the **Print** menu option, and the dialog box is the same one that is displayed from other applications.

Print

Provides standard printing features for KMP titles. This option is not yet implemented for OS/2.

This menu command calls the `printTitle` method on the title container that has user focus (`theTitleContainer`). The default implementation of `printTitle` calls the `printWindow` method on the frontmost window of the title (`windows[1]`). The default implementation of `printWindow` does nothing. You can create a subclass of `TitleContainer` and specialize the `printTitle` method, or you can create a subclass of `Window` and specialize the `printWindow` method to provide an implementation of printing for your title.

Quit or Exit

Quits ScriptX, and, by default, does not save any titles, accessories, or libraries previously opened. This menu command is named **Quit** on the Macintosh and **Exit** in Microsoft Windows and OS/2.

This menu command calls the `quit()` global function:

```
quit()
```

About ScriptX

Provides version and copyright information about ScriptX. On the Macintosh, this menu item is located as the first item under the Apple menu.

The Edit Menu

Provides standard editing features for both text in the ScriptX Listener and objects in ScriptX titles. The target of all commands in the Edit menu is the ScriptX window that currently has user focus. This window may belong to a ScriptX title or it may be the ScriptX Listener window.

Within the ScriptX Listener, you can cut, copy, and paste text. For example, you can type a ScriptX statement in the Listener and execute it, then copy it from the Listener window and paste it into the text file for a script.

Within a title, the title must implement cut, copy, and paste features. See the chapter “Title Management” in the *ScriptX Architecture and Components Guide* for more on implementing editing features in a title.

Undo

Reverses the effect of the previous edit in the Listener window.

Note – ScriptX does not provide any API to make use of the **Undo** menu command in a title.

Cut

When the Listener window is active, this menu command cuts the current selection in that window, which removes it from the window and places it on the clipboard. The clipboard’s `typeList` instance variable is set to `@text`.

When a window belonging to a title container is active, this menu command calls the `cut` method on `theTitleContainer` (which in turn calls `cut` on its frontmost window, which has no default implementation):

```
cut theTitleContainer
```

Copy

When the Listener window is active, this menu command copies the current selection in that window, which leaves it in the window and places a copy of it on the clipboard. The clipboard’s `typeList` instance variable is set to `@text`.

When a window belonging to a title container is active, this menu command calls the `copy` method on `theTitleContainer` (which in turn calls `copy` on its frontmost window, which has no default implementation):

```
copy theTitleContainer
```

Paste

When the Listener window is active, this menu command pastes the text from the clipboard into that window at the insertion point.

When a window belonging to a title container is active, this menu command calls the `paste` method on `theTitleContainer` (which in turn calls `paste` on its frontmost window, which has no default implementation):

```
paste theTitleContainer
```

Clear

When the Listener window is active, this menu command deletes the currently selected text, which removes it from the window. Has no effect on the clipboard.

When a window belonging to a title container is active, this menu command calls the `clear` method on `theTitleContainer` (which in turn calls `clear` on its frontmost window, which has no default implementation):

```
paste theTitleContainer
```

Select All

Selects all text in the Listener window. (Not available in the KMP.)

Fonts (Windows only)

Changes the font, style and size of text in the Listener window. The font you choose is saved for future sessions. (Not available in the KMP.)

The Window Menu

This menu provides a list of all main windows currently open in ScriptX, listing them by the title they were given when created in ScriptX. If the ScriptX Listener window is open, it is included in this menu. Selecting a window from this menu makes it the key window and gives it user focus.

In the KMP, this menu is empty until you open a title that has a window.

In Microsoft Windows, this menu contains three extra commands: **Cascade**, **Tile**, and **Arrange Icons**. These are standard menu items that are described in the Microsoft Windows documentation.

The Tools Menu

This menu does not appear unless the Tools directory is present at the same level as the ScriptX executable file. This menu provides a list of all tools currently available in ScriptX, as well as high-level controls for any tool that's currently selected.

This menu is divided into two parts by a horizontal line. The upper part includes **About**, **Preferences**, and **Quit**. These commands are dimmed until you choose a tool from the lower part of this menu. The top three menu commands then apply to the tool that is active.

About

Provides version and copyright information about the currently selected tool. (The information may be displayed in the Listener window.)

Preferences

Lets you set preferences for the currently selected tool.

Quit

Quits the currently selected tool.

Components

Displays components that enhance ScriptX. Examples include Grapher, which displays a class tree of all classes, and RGB Color, a tool for editing color. All components are currently not supported.

Browsers

Displays browsers available in ScriptX. Currently, these are the Class, Instance, and Source browsers described elsewhere in this document.

Debugger

Displays the debugger available in ScriptX.

The *ToolName* Menu

In this place appears the Instances, Modules, Source, or Debugger menu, corresponding to the tool that selected from the Tools menu that is currently active. This menu displays a list of menu commands specific to that tool.

CHAPTER

8

The Browser

The ScriptX Browser Component provides a visual interface that enables you to view modules, classes, instances and source code in the ScriptX development environment.

You can use the browser to see what classes, global variables, functions and methods are contained in a module. You can also change the value of global variables.

You can see the subclasses and superclasses of a class, the instance variables defined on a class, the methods defined on a class, and all instances of a class. You can also change the values of class variables.

You can see and change the values of the instance variables of an instance.

For collections, you can view the contents of the collection.

You can view and edit the definitions of functions and free methods defined in scripts that you have loaded, and also view the definition of methods that have been specialized at the instance level.

Loading the Browser

You can load the ScriptX Browser anytime during a ScriptX session so long as the files `widgets.sxl` and `toolutil.sxl` are in the startup directory or a directory in the startup directory. To load the Browser, open the title container `browser.sxt`, which will most likely be located in a folder called `Tools` or `Toolsx` in the startup directory.

If you want the Browser to load automatically when ScriptX starts up, put the file `browser.sxt` in a folder called `Tools` in the startup directory.

After the Browser loads, whether you load it explicitly or it loads when ScriptX starts up, the browsers remain closed until you open them.

To display the **Browser** menu, which has commands that allow you to interact with the browsers, select the **Browser** command from the **Tools** menu. You will see the **Browser** menu in the menu bar.

Using the Browser

To display a module, class, or instance, in the appropriate browser, call the `inspect` method on it from the Listener:

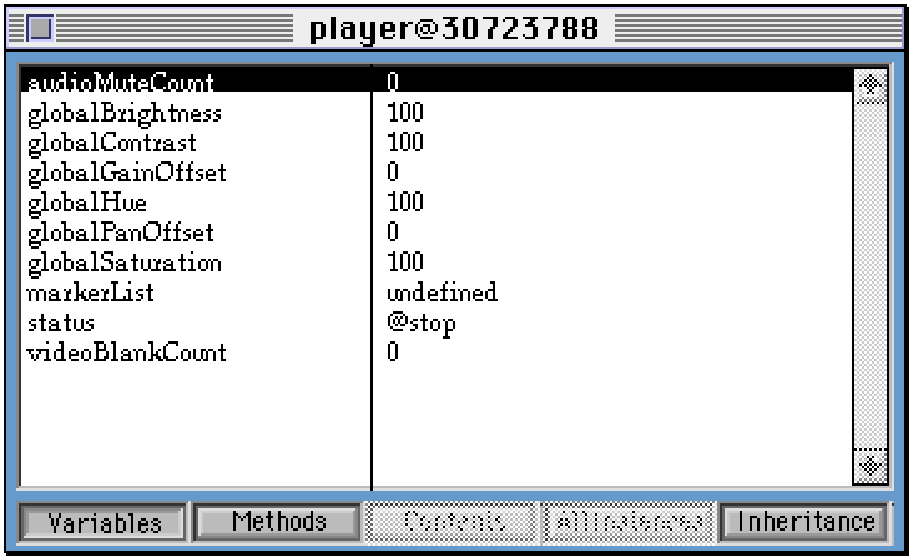
```
global p: = new player

-- Display p in a browser
inspect p

-- Display the class Player in a Browser
inspect player

-- Display the contents of the module myModule in a browser
inspect (getModule @myModule)
```

Figure 8-1 shows the Browser displaying a `Player` object.



player@30723788

audioMuteCount	0
globalBrightness	100
globalContrast	100
globalGainOffset	0
globalHue	100
globalPanOffset	0
globalSaturation	100
markerList	undefined
status	@stop
videoBlankCount	0

Figure 8-1: The Browser

You can cause an item listed in one browser window to be displayed in its own browser window by double-clicking the item. If you hold the Shift key down while double-clicking, the current browser window updates to show the selected item, instead of opening a new browser window.

Resizing the Browser

You can change the size and shape of a browser window by grabbing the bottom right hand corner with the mouse, and dragging it to the desired shape.

You can move the panel divider in the browser by clicking on it to grab it, and moving it left of right.

Stepping Back to the Previous Browser Display

To step back to the display of the previously displayed object, or module in a browser, select the **Back** command in the **Browser** menu. If you continue selecting the **Back** command, you eventually cycle back to the originally displayed item.

Note – If you display an object of any kind in a browser, the object is added to the browser's history list, which means it can't be purged from memory until it comes off the history list. Each browser keeps track of approximaltely 20 objects in its history list.

Updating Browsers

If the state of an object displayed in a browser changes, the browser does not update automatically. To update the browser to show the latest state of the displayed item, choose the **Refresh** menu command from the menu for that kind of Browser.

For example, if the a browser window is displaying the object `p` (which is a `Player` instance), and you change the value of `p`'s `time` instance variable in the Listener, the browser does not update to reflect the change. To update the browser, click on it to ensure that it is the top browser and select the **Refresh** command in the **Browser** menu.

If you use the **Edit** command in the **Browser** menu to change the value of a variable or an item in a collection, then the browser updates to show the change.

What You See in the Browser

The title of a browser windows indicates what class, object, or module is currently displayed. Each browser window shows two panes, and five buttons.

The buttons are:

- **Variables**

  Shows the instance and class variables (if applicable) for the displayed object.

  The left hand pane shows the method name, the right hand pane shows its value.

- **Methods**

  Shows the instance and class methods (if applicable) for the displayed object.

  The left hand pane shows the variable name, the right hand pane shows its its full name, including which module it is in.

- **Contents**

  Shows the contents of the displayed object, if it is a collection or a module.

  The left hand pane shows the key, the right hand pane shows the value.

- **AllInstances**

  Shows all instances of the displayed class.

  The left hand pane shows the number of the object, the right hand pane shows the object.

- **Inheritance**

  Shows the superclasses and subclasses (if applicable) for the displayed object.

  The left hand pane shows the names of the superclasses and subclasses, the right hand pane shows the full name including the module.

Depending on what is being displayed, one or more of these buttons may be grayed out if they are not applicable. For example, when displaying a non-collection instance, the **Contents** button is not useful so it is grayed out. When displaying an instance, the **All Instance** button is grayed out.

Use these buttons, in conjunction with the commands in the **Browser** menu, to determine what is displayed in the browser.

The menu commands that help determine what is displayed are:

- **Sort Order**

  Determines whether the displayed items are displayed alphabetically (**Alphabetical**) or sorted by inheritance (**Inheritance**).

  Default value is **Alphabetical**.

- **Select IVs by**

When instance and class variables are displayed, this command determines hether actual instance variables (**Slot name**) or virtual instance variables (**Virtual Name**) are displayed.

Actual instance variables are defined in the class definition, whereas virtual instance variables are implemented only through setter and getter methods.

Default value is **Slot Name**.

- **Specialization Level**

  Determines whether the displayed items include the instance variables or methods defined on the local class (**Class**) or includes those defined in all superclasses (**All**).

  Default value is **Class**.

- **Include Accessor Methods**

  When methods are displayed, determines whether setter and getter methods are also displayed.

  By default, accessor methods are not included.

- **Include Root Object**

  Determines whether instance variables and methods inherited from the class **RootObject** are displayed.

  By default, instance variables and methods inherited from **RootObject** are not included.

Sample Uses

What you see in the browser depends on what button and what menu command are currently selected.

For example, to see the value of the `presentedBy` instance variable of a `TwoDShape` instance, you need to ensure that the **Variables** button is selected, and the **Specialization level** is set to **All**, since `presentedBy` is inherited from `Window` and is not directly defined on `TwoDShape`.

Suppose you define a class that has an instance variable `FavFood` that is implemented through setter and getter methods, and is not declared in the class definition. To see this method displayed, ensure that the **Methods** button is selected, and the **Select IVs by** command is set to **Virtual Name**.

Editing Values in the Browser

To change the value of an instance variable displayed in a browser window, click on the instance variable entry to select it, then choose **Edit** from the **Browser** menu.

To change the value of an element in a collection displayed ina browser, click on the desired element to select it, the choose **Edit** from the **Browser** menu.

In the dialog box that appears, enter the new value, then press the **OK** button. The dialog box disappears, and the browser window updates to show the new value.

Editing Methods and Functions

You can change the source code for user-defined functions and free methods in the browser so long as the source code is available. The source code will be available if the free method or function was defined in the Listener in the current ScriptX session, or it was defined in a file that was loaded into ScriptX either by the **Open** menu command, or by using the `fileIn` method with the `debugInfo` keyword set to `true` (which is the default setting.)

To edit the definition of a function, display the contents of the module containing the function. Select the desired function, then choose **Edit** from the **Browser** menu. In the window that appears, make your changes then select the **OK** button. The changes will be saved.

To edit the definition of a method, this, ensure that the `Methods` button is selected so that the methods are displayed. Select the desired method, then choose **Edit** from the **Browser** menu. In the window that appears, make your changes then select the **OK** button. The changes will be saved.

A free method is a method that is defined outside of a class definition. For example, the methods `printFood` and `printName`, defined below, are both instance methods on the class `PartyGoer`. However, `printFood` is defined within the class definition, so its definition will never be visible in the Class Browser. The method `printName` on the other hand is a free method and its definition can be viewed in the browser.

```
class PartyGoer (RootObject)
instance variables
   food
   name
instance methods
      method printFood self ->
         (format debug "My favorite food is %*" self.food @normal)
end

method printName self {class PartyGoer} ->
(format debug "My name is %*" self.name @unadorned)
```

More On Actual Versus Virtual Instance Variables

The browser can show the actual instance variables or the virtual instance variables. Actual instance variables are those that are defined in the class definition. Virtual instance variables are those that are defined by setter and getter methods.

The following code shows a simple class definition. In this example, `pricePerGram` and `weight` are actual instance variables, since they are defined in the class definition, whereas `cost` is a virtual instance variable, since it is implemented through a setter and getter method.

```
class Fruit (RootObject)
instance variables
   pricePerGram
   weight
end

method costSetter {class Fruit} value ->
(  "Cannot change the cost directly"
)

method costGetter {class Fruit} ->
(  pricePerGram * weight
)
```

Use the **Select IVs by** menu command in the **Browser** menu to select whether to show actual or virtual instance variables. The **Slot Names** choice shows actual instance variables. The **Virtual Names** choice shows instance variables implemented through seter and getter methods.

Note that for scripted classes, all actual instance variables also have setter and getter methods, so the list of virtual instance variables also includes the actual instance variables.

For ScriptX Core classes however, actual instance variables do not necessarily have setter and getter methods, so the list of virtual instance variables does not necessarily include all the actual instance variables too. However, if a class defines a setter and getter method for an inherited instance variable, that instance variable shows up in both the actual and virtual instance variable lists.

Note that a browser window displays and updates quicker when displaying actual instance variables than when displaying virtual instance variables.

Using the Picklist

You can also create your own list of objects to browse. This list is known as the Pick List.

To create a Pick List, select the **Pick List** command from the **Browser** menu.

A browser opens. One (possibly the only) entry in the left hand column is **<new category>**.

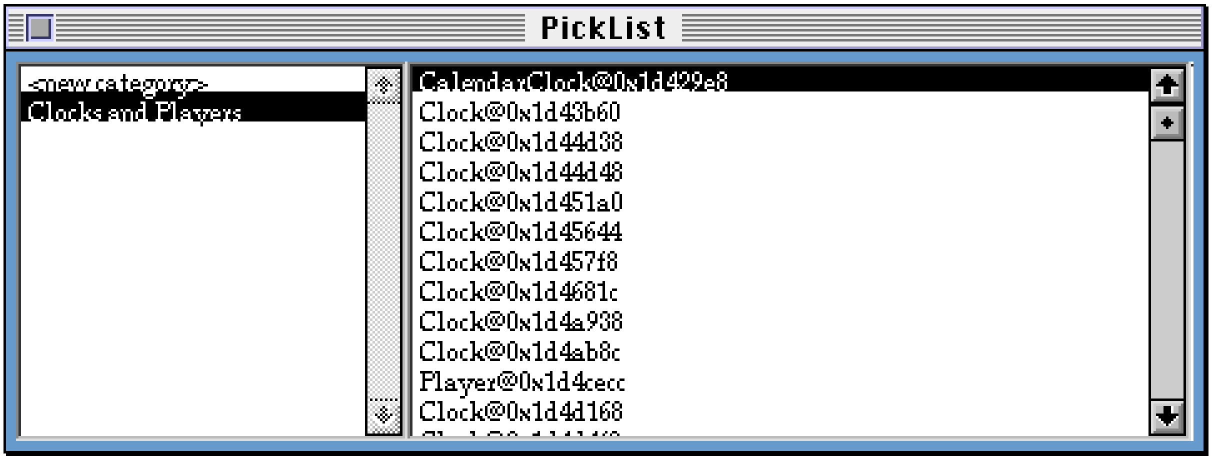
Select **<new category>**. In the dialog box that opens, enter the name of your category, for example, `Clocks and Players`, (don’t use quotes) then press **OK**.

In the next window that appears, enter a function that returns a collection of objects to browse, for example `(-> allInstances clock)`. Then press **OK**.

The PickList browser updates to show the category heading on the left, and the objects returned by the function on the right, as illustrated in Figure 8-2.

You can click on any object in the right hand column of the Pick List to browse it.

The picklist is saved in the file `picklist.ini`, which you can add directly in any text editor to delete or add entries as well.



<new category>
Clocks and Players

CalendarClock@0x1d429e8
Clock@0x1d43b60
Clock@0x1d44d38
Clock@0x1d44d48
Clock@0x1d451a0
Clock@0x1d45644
Clock@0x1d457f8
Clock@0x1d4681c
Clock@0x1d4a938
Clock@0x1d4ab8c
Player@0x1d4cecc
Clock@0x1d4d168

Figure 8-2: The PickList

Further examples of useful categories and their functions are:

Category is **Current spaces**

Function is:
```
( -> allInstances Space)
```

Category is **Current modules**

Function is:
```
( -> allInstances ModuleClass)
```

Category is **All scripted classes**

Function is:
```
( -> chooseAll (allInstances RootClass) (cl z -> sxdefined cl) 0)
```

Reference

This section provides a reference to the browser and its menus.

Tools Menu

The **Tools** menu includes commands that are relevant to all tools. It also allows you to select which additional menu to display.

About

Displays information about the currently selected tool.

Preferences

Opens a dialog box that lets you set preferences for the browser, so long as the Browser is the currently selected tool. (To make it be the currently selected tool, either click on a browser or select the **Browser** menu command in the **Tools** menu.)

You can set the following:

- the number of items the browser keeps in its history list
- the default setting for the **Specialization** menu command
- the default setting for the **Sort IVs** by menu command
- the default font for text in the browser
- the default size for text in the browser
- the default color for for the browser background (blue might be the only choice)

If you make changes and press **OK**, the changes are saved for the current session only. If you make changes and press **Save**, the changes take effect immediately, and are also written out to a preferences file (`browser.ini`). When you start ScriptX up subsequently, the preferences are read back in and continue to take effect.

Quit

Closes the currently selected tool, removes its menu from the menu bar and removes it from the list of tools available through the **Tools** menu.

Browser

This command causes the **Browser** menu to appear in the menu bar.

Browser Menu

The **Browser** menu appears when the browser is the currently selected tool.

Sort Order

Lets you choose how to sort instance/class variables and methods. The choices are:

- **Alphabetical**

Sorts alphabetically.

- **Inheritance**

  Sorts the methods and instance/class variables according to which class they are inherited from.

Select IVs by

Lets you choose whether to display actual or virtual instance/class variables. The choices are:

- **Slot Name**

  Displays actual instance/class variables (that is, instance/class variables that are defined in the class definition.)

  Displays all methods, including setter and getter methods.

- **Virtual Name**

  Displays virtual instance/class variables (that is, instance/class variables that are defined by setter and getter methods.)

  Displays methods excluding methods for which there is a getter method, since the corresponding instance/class variables appears in the list of variables.

Specialization Level

Let you decide whether local or inherited instance/class variables and methods should be displayed. The choices are"

- **Class**

  The instance/class variables and methods that are defined on or specialized for the displayed instance or class are listed. If an instance is displayed, the methods that are defined on or specialized for its direct class parent are also displayed.

- **All**

  All instance/class variables and methods that are available to the displayed instance or class are listed. This includes locally defined ones, and all the ones inherited from the parent class and all superclasses.

Include Accessor Methods

Determines whether or not accessor methods (that is, setter and getter methods) are included in method listings. The default is no.

Include Root Object

Determines whether methods and instance or class variables inherited from the class `RootObject` are included in listings. The default is no.

Picklist

Displays a customizable list of browsable objects. For details see “Using the Picklist” on page 88.

Set Breakpoint

Lets you set a breakpoint for a method or function that is written in ScriptX. (You cannot set breakpoints on methods defined on ScriptX Core Classes, since they are written in OIC.)

Back

The **Back** menu sets the target object of the browser to the previous item inthe browser's history list.

Edit

Lets you change the values of instance/class variables, change elements in a collection, and edit function or method definitions.

If the selected value is editable, this menu command opens a dialog box that you can type a new value into. Selecting **Change** in the dialog box saves the change and updates the browser display to show the change.

Refresh

Updates all open browser windows to reflect any changes that have occurred since they were opened or refreshed.

CHAPTER

9

Debugger

The ScriptX Debugger is a tool that helps you to debug code written in the ScriptX language. You can use the debugger to investigate errors that occurred, or to execute code one step at a time to see what happens.

When an exception occurs, you can use the debugger to try to find out what caused the exception. You can view the sequence of function or method calls that lead up to the exception. (This sequence is known as the call stack.) You can see what arguments each function or method in the stack was called with, and you can see the code for each function or method.

You can use the debugger to specify breakpoints, which are functions or methods that cause your code to be suspended and the debugger to become active. When the debugger becomes active, you can step through the ScriptX code one expression at a time to see what happens and what changes at each step.

You can also set watchpoints, which are expressions that the debugger constantly monitors. Watchpoints show the current value of a variable or an expression.

The debugger is a tool for developers. It is available in the ScriptX development environment, but not in the Kaleida Media Player. Use the debugger to aid in the development of your title, but do not write a title so that it depends on the debugger in any way at runtime.

How to Load The Debugger

You can load the debugger anytime during a ScriptX session so long as the files `widgets.sxl` and `toolutil.sxl` are in the startup directory. To load the debugger, open the title container `debugger.sxt`which will most likely be located in a folder called `Tools` or `sxTools` in the startup directory.

If you want the debugger to load automatically when ScriptX starts up, put the file `debugger.sxt` in a folder called `Tools` in the startup directory.

To display the **Debugger** menu, which has commands that allow you to open and use the debugger, select the **Debugger** command from the **Tools** menu. You will see the **Debugger** menu in the menu bar.

You can have one debugger window per thread.

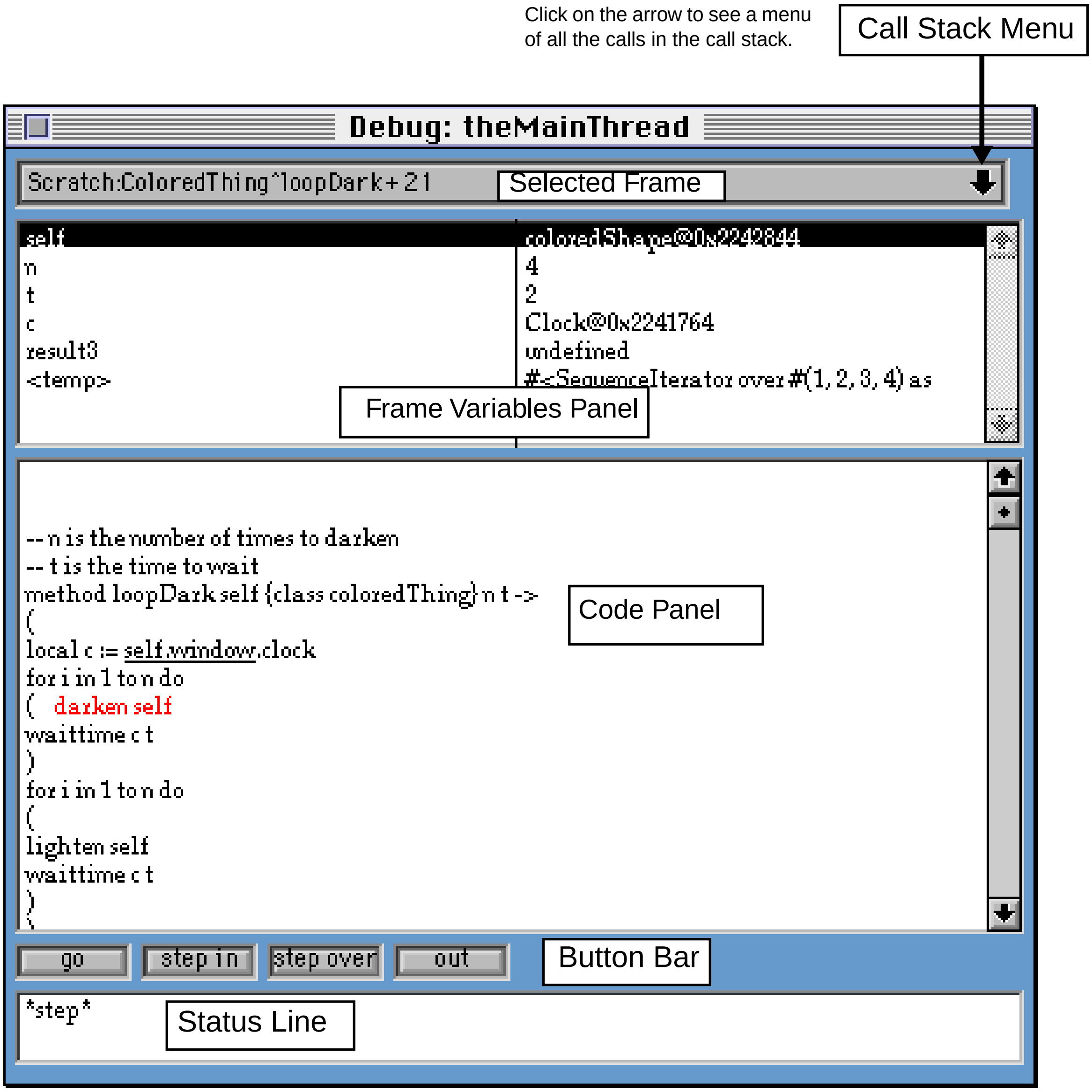
Opening the Debugger

If the **New Debugger on Exception** flag is set to to true (which is the default setting), a debugger will open automatically when an exception occurs. (Note that the debugger does not open if a system error occurs.)

To change the **New Debugger on Exception** preference setting, ensure that the **Debugger** menu is visible in the menu bar, then select the **Preferences** command from the **Tools** menu. Set the preferences as you desire. See “Preferences” on page 111 for more information on setting preferences.

You can cause the debugger to open when a method or function executes by setting a breakpoint for the desired method or function. See “Setting Breakpoints” on page 103 for detail on how to set breakpoints. Or you can recompile your method or function to include a call to `break ()`, which will open the debugger when that point is reached.

You can also open the debugger for a particular thread by selecting `Interrupt` from the Debugger menu and choosing the appropriate thread, or by typing `break ()` in the Listener.


Click on the arrow to see a menu of all the calls in the call stack.

self	coloredShape@0x2242844
n	4
t	2
c	Clock@0x2241764
result3	undefined
<temp>	#<SequenceIterator over #(1, 2, 3, 4) as

```
-- n is the number of times to darken
-- t is the time to wait
method loopDark self {class coloredThing} n t ->
(
local c := self.window.clock
for i in 1 to n do
(  darken self
waittime c t
)
for i in 1 to n do
(
lighten self
waittime c t
)
```

```
*step*
```

Figure 9-1: The Debugger Window

Resizing the Debugger

To change the size or shape of the debugger, grab the bottom right corner with the mouse and drag it to the desired shape. You can also drag the divider bar in the Frame Variables panel to the left or right to resize the parts of that panel.

Description of the Debugger

The Debugger Window is illustrated in Figure 9-1. It has the following components:

- Selected Frame Field
- Call Stack Menu
- Frame Arguments Panel
- Code Panel
- Button Bar
- Status Line

You can change the size and shape of the Debugger Window by grabbing the bottom right corner with the mouse and dragging the mouse to make the window the desired size.

Figure 9-1 shows the Debugger Window.

Selected Frame Field and Call Stack Menu

The Selected Frame Field shows the currently selected frame, which is one of the calls in the call stack.

To view all the calls in the call stack, click on the arrow at the right end of the Selected Frame Field. A menu appears listing all the calls in the stack. Selecting on of these calls causes the Selected Frame Field to show the selected call (or frame). The Frame Variables panel updates to show the arguments that the function or method was called with, and also the local values pushed on the stack during the call.

The Code Panel updates to show the code for the selected frame. See “Selected Frame Field and Call Stack Menu” on page 97 for more information about the code listing in the Code Panel.

Call Stacks

The Call Stack Menu lists the functions or methods that have been invoked in the current path of execution.

For example, suppose the following functions are defined:

```
function functionA -> (functionB())

function functionB -> ( functionC())
```

When you call `functionA`, it calls `functionB`, which calls `functionC`. When `functionC` is reached, the call stack is:

```
functionC
functionB
functionA
```

Only the function and method calls that lie directly in the path that leads to the current call are included in the call stack. For example, suppose `functionA`, `functionB` and `functionC` are defined as follows:

```
function functionA -> (
   functionA1()
   functionB())
```

```
function functionB -> (functionC())
```

When `functionC` is reached, the call stack does not include `functionA1`, since it was not involved at any level in invoking `functionC`.

Each entry in the call stack menu has the following format:

module:callName offset

where *module* is the module containing the function or method; *callName* is the function or method that was called; and *offset* is the number of the bytecode instruction that was reached. (Each ScriptX expression is compiled into bytecode instructions.)

For example:

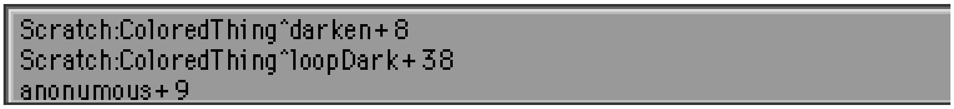
```
Scratch:functionB+3
```

which means `functionB` in the scratch module at the instruction starting at the third byte. In most cases, the only interesting information conveyed by the offset is whether it is 0 or greater than 0. If it is 0, that means you have just entered the frame and have not yet executed anything in the frame. If the offset is greater than 0, that means that you are part way through executing the frame.

Expressions that were entered from the Listener show up in the Call Stack Menu as `anonymous`. For example, Figure 9-2 shows the Call Stack Menu after entering the following function call in the Listener:

```
loopDark sun 20 2 ()
```

The expression that was entered in the Listener shows up initially in the Call Stack Menu simply as `anonymous`+0.



```
Scratch:ColoredThing^darken+8
Scratch:ColoredThing^loopDark+38
anonymous+9
```

Figure 9-2: The Call Stack Menu showing an anonymous call

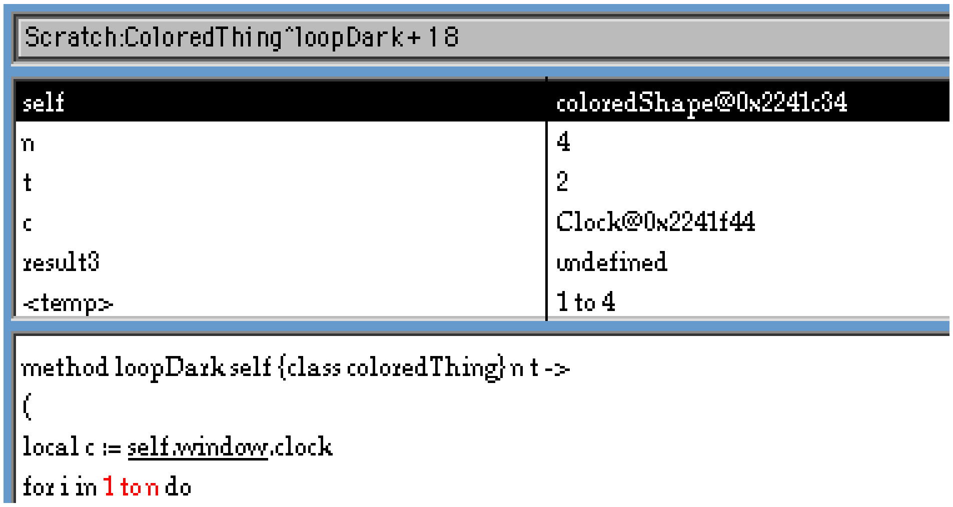
Frame Variables Panel

The Frame Variables panel shows the arguments and local stack values appropriate to the selected frame in the Call Stack Menu. A frame is a specific invocation of a function or method.

When you are executing code step by step, the selected frame is always displayed in the Call Stack Field. As you step through code, the Frame Variables panel updates to show the local stack values as they change.

For example, Figure 9-3 shows the Frame Variables panel as it appears at one point during the execution of the `loopDark` function.

You see that `loopDark` method was called with *self* as a `ColoredShape` object; *n* as 4 and *t* as 2. So far, the variable *c* has been bound to a `Clock` object, *result* (a name that the debugger uses for results) is bound to the `undefined` object, and a temporary variable holds the value `1 to 4`. which corresponds to `1 to n` in the `for` statement.



Scratch:ColoredThing^loopDark+18

self	coloredShape@0x2241c34
n	4
t	2
c	Clock@0x2241f44
result3	undefined
<temp>	1 to 4

```
method loopDark self {class coloredThing} n t ->
(
local c := self.window.clock
for i in 1 to n do
```

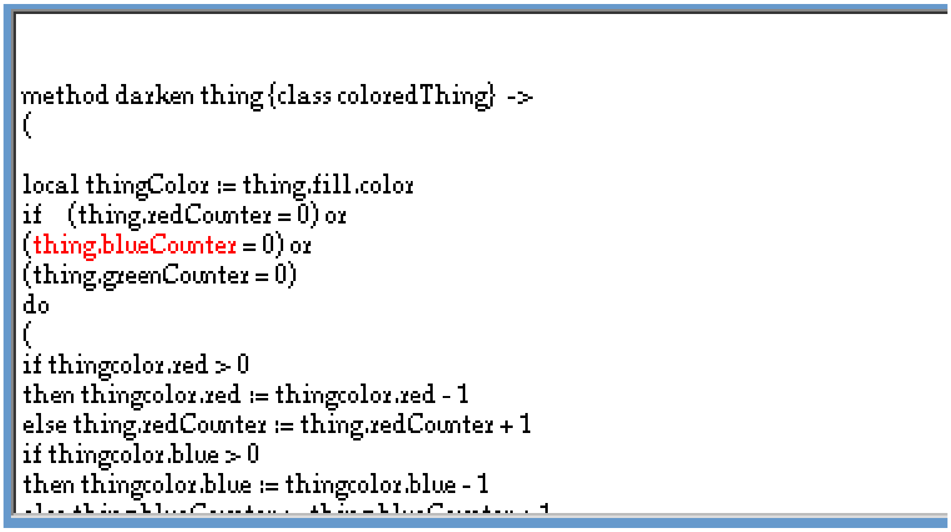
Figure 9-3: Frame Variables panel shows the arguments and stack values of a frame

If you double click on an object in the Frame Variables panel, the Browser will open to display that object, so long as the Browser is loaded. See the Browser Chapter 8 for information about the Browser.

Note also that you can grab the divider bar with the mouse and move it to the left or right.

Code Panel

The Code Panel shows the ScriptX source code that makes up the selected fame in the Current Frame Field. In Figure 9-4 the Code Panel shows the definition for the function `darken`.



```
method darken thing {class coloredThing} ->
(

local thingColor := thing.fill.color
if   (thing.redCounter = 0) or
(thing.blueCounter = 0) or
(thing.greenCounter = 0)
do
(
if thingcolor.red > 0
then thingcolor.red := thingcolor.red - 1
else thing.redCounter := thing.redCounter + 1
if thingcolor.blue > 0
then thingcolor.blue := thingcolor.blue - 1
```

Figure 9-4: The Code Panel showing the source for the `darken` method

The Code Panel can show source code for functions or methods that were loaded with their source code, as discussed in “Accessing Source Code for Methods and Functions” on page 107.

Underlined code indicates a break point.

You can use the **Step Into** and **Step Over** buttons to execute one ScriptX experssion at a step. You can use the **Out** button to execute the current function or method to completion.

You can also set breakpoints in the Code panel by pressing the mouse down on the desired expression, and choosing **Set Breakpoint** from the menu that appears. Similarly to remove a breakpoint, you can mouse down on underlined code in the Code Panel, and choose **Remove Breakpoint** from the menu that appears.

Stepping Through Code

You can use the debugger to set breakpoints, which suspend your program. When a breakpoint is reached, you can step through the program one expression at a time.

You might want to set a breakpoint, for example, on a function or method that causes an exception. Or you might want to set a breakpoint to stop a program at a place just before it starts behaving differently to how you expect it to behave. Then you can execute the program one expression at a time to see what changes occur at each step.

You can set multiple breakpoints, so that you can step through a few instructions, and then press the **Go** button to continue running until the next breakpoint.

If an exception occurs while stepping through a program, you will no longer be able to continue stepping. In this case, use the **Go** button to continue running the program and leave the debugger.

As you step through code, the expression currently being executed is highlighted in the Code Panel (so long as the source code is loaded.)

Entering the Debugger Immediately

You can also enter the debugger by selecting the **Interrupt** menu command from the **Debugger** menu and choosing the desired thread to interrupt. (Note that the Listener uses the main thread.) If a program is currently running, it stops and invokes the debugger. If a program is not currently running, the very next ScriptX expression to be executed in the chosen thread invokes the debugger.

So if you interrupt the main thread, the next time you evaluate an exression in the Listener, the debugger opens and you can use it step through the code in the expression.

Note that although you can interrupt Core (OIC) code, you will not see any interesting information in the debugger window.

You can also use the break function to force a break. For example, if you enter the following code in the Listener, the `break` function invokes the debugger, and then you can step through the code in `myFn`.

```
(break (); myFn a b c)
```

Stepping Through Code

When stepping through code in the debugger, you can use the **Step Into, Step Over,** and **Out** buttons. Both **Step Into** and **Step Over** execute the next ScriptX expression in the Code Panel. (This will be the highlighted instruction.) The **Out** button executes all the remaining expressions in the currently displayed function or method.

The **Step Into** button executes the currently highlighted expression. If that instruction calls a function or method written in ScriptX, then the Code Panel updates to show the code for the called function or method. (You can use the keyboard accelarator F8 instead of the **Step Into** button.)

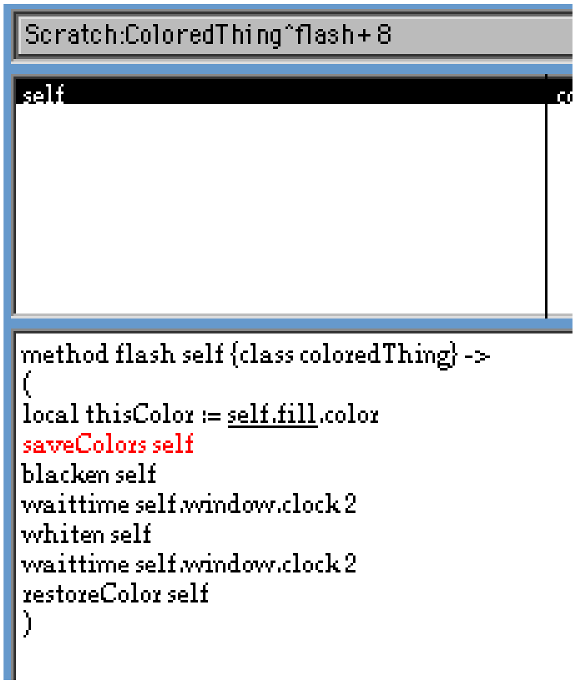
The **Step Over** button executes the currently highlighted expression to completion. If that expression calls a function or method written in ScriptX, the entirre function or method is run to completion without stepping. (You can use the keyboard accelarator F9 instead of the **Step Over** button.) Note that you can step over function or method calls, but you cannot step over statements. Each expression in a statement is executed individually, regardless of whether you step into it or step over it.

In the top debugger shown in Figure 9-5 the next thing to be executed is the `saveColors` method. The debugger on the bottom left shows the results if you **Step Into** into the `saveColors` function. The debugger on the bottom right shows the results if you **Step Over** the `saveColors` function.

The **Out** button executes all the expressions in the current function or method that have not been executed yet. It executes them to completion, and effectively steps you back out to the routine that called this function or method. You can use the F6 key instead of the **Out** button.

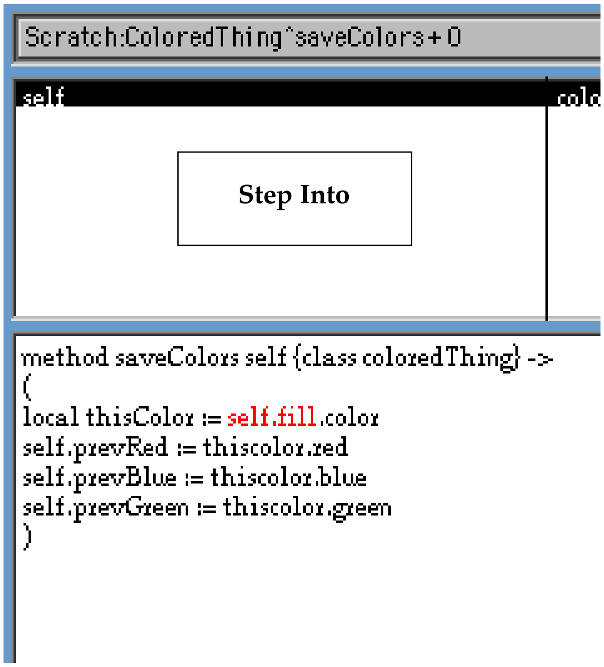
Go

When you press the **Go** button, the system continues running as it would in the absence of the debugger. When an exception occurs, the usual behavior of the system is to return to the nearest guard clause. If there is no intervening guard clause, the Listener prints a message about the exception, and the current thread stops. (Note that the F5 keyboard accelerator is bound to the **Go** button.)

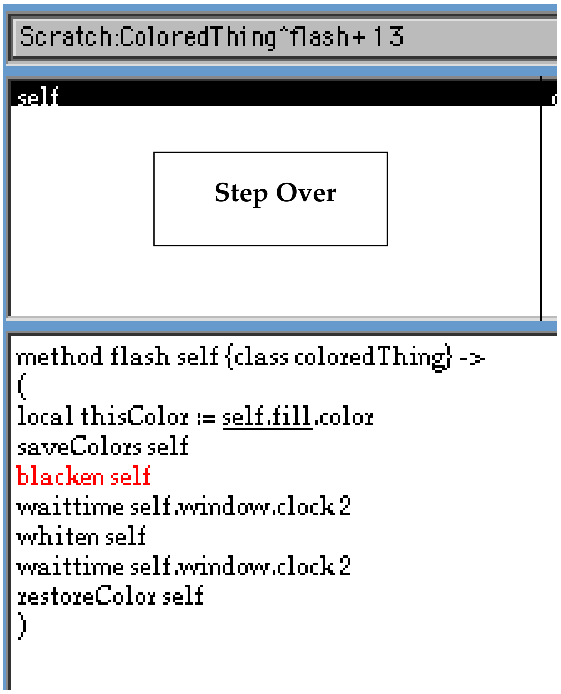


```
method flash self {class coloredThing} ->
(
local thisColor := self.fill.color
saveColors self
blacken self
waittime self.window.clock 2
whiten self
waittime self.window.clock 2
restoreColor self
)
```

The debugger is ready to execute the saveColors method.



```
method saveColors self {class coloredThing} ->
(
local thisColor := self.fill.color
self.prevRed := thiscolor.red
self.prevBlue := thiscolor.blue
self.prevGreen := thiscolor.green
)
```



```
method flash self {class coloredThing} ->
(
local thisColor := self.fill.color
saveColors self
blacken self
waittime self.window.clock 2
whiten self
waittime self.window.clock 2
restoreColor self
)
```

Figure 9-5: Difference between Step Into and Step Over

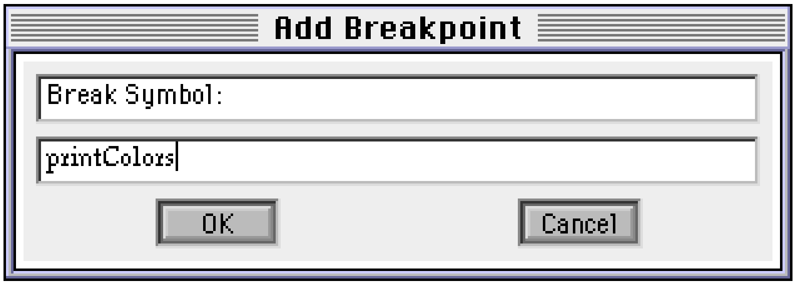
Setting Breakpoints

You can set a breakpoint in several ways.

From the **Debugger** menu, choose **Show** then choose **Breakpoint** to open a Breakpoints window. In the window, specify the ScriptX function or method name for which to set the breakpoint.

This actually sets the breakpoint on the first expression within that function or method. You can set breakpoints for user-defined or dynamically loaded functions and methods written in the ScriptX language. You cannot set breakpoints on anything other than functions and methods that have been defined in the ScriptX language, which means you cannot set breakpoints on methods on ScriptX core classes, since they are written in OIC not ScriptX. It also means you cannot set breakpoints on functions in loadable C code.

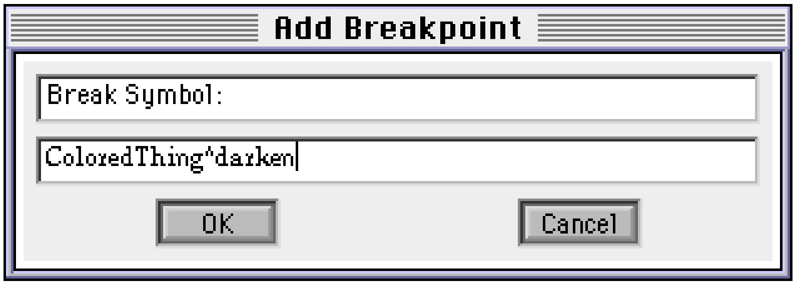
For a function, enter the name of the function. For example:



Break Symbol:

printColors

For a method, the entry should be in the form *class*^*method*. For example, to set a breakpoint on the `darken` method of the class `ColoredThing`, the entry would be: `ColoredThing^darken`. For example:



Break Symbol:

ColoredThing^darken

If you set a breakpoint for a method, the debugger will be invoked when the method is called on any instance of the class or its subclasses (and their subclasses and so on.)

You can specify breakpoints for methods defined on a class. You cannot specify breakpoints for method for a class that inherits the method.

You can also set breakpoint by using the **Set Breakpoint** command in the Class Browser.

In the class browser, select the desired method and select the **Set Breakpoint** command from the **Classes** menu. You can only do this for user-defined methods, not for ScriptX core classes method. The reason for this limitation is that ScriptX core classes are written in OIC, not in ScriptX.

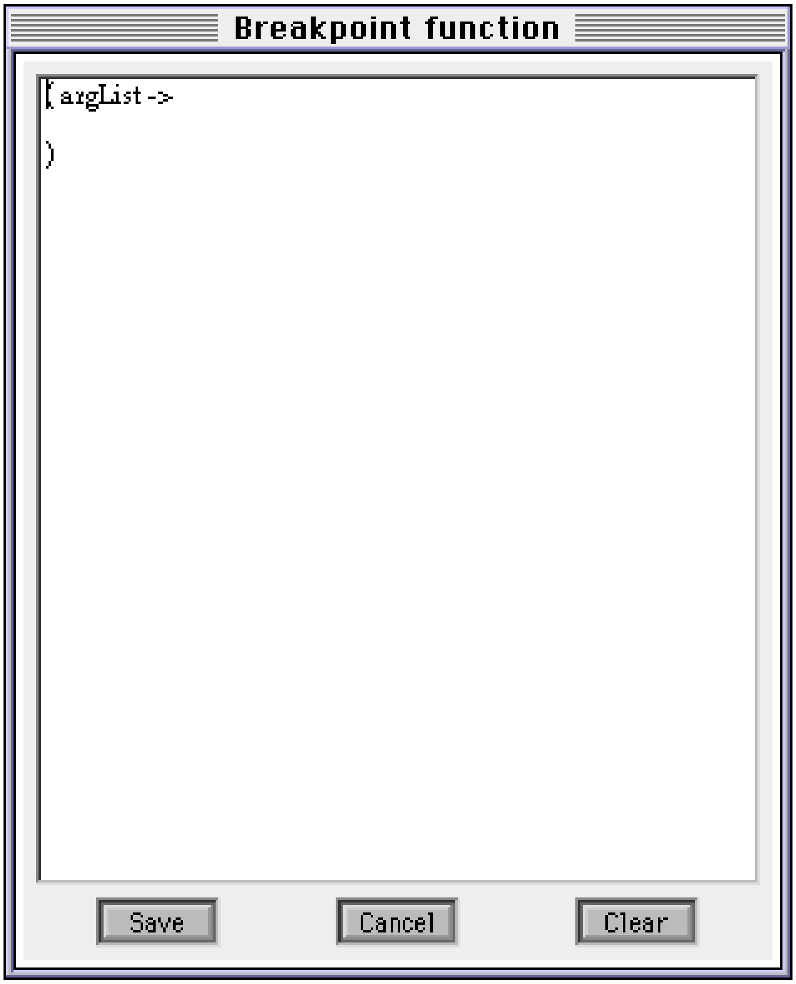
You can also set breakpoints in the Code panel by pressing the mouse down on the desired expression, and choosing **Set Breakpoint** from the menu that appears. Similarly to remove a breakpoint, you can mouse down on underlined code in the Code Panel, and choose **Remove Breakpoint** from the menu that appears.

Note – If you recompile a function or method that has breakpoints, those breakpoints will be lost, even though they will still appear in the breakpoints window.

Defining a Function to Run at a Breakpoint

You can specify a function to be called when a breakpoint is encountered. To set such a function, select the desired breakpoint in the Breakpoints window and click the **Edit** button.

The Edit Breakpoint window opens, showing a skeletal anonymous function whose argument is `argList`, (as illustrated in Figure 9-6). The `arglist` argument is an array of the arguments passed to the function or method on which the breakpoint is set. Define the function as you like (but do not edit the argument list.) To compile the function, simply close the Edit Breakpoint window.



```
( argList ->

)
```

Figure 9-6: The Edit Breakpoints window when it opens

In the Breakpoints window, breakpoints that have an attached function are indicated by an asterisk (*).

When the breakpoint is encountered while your code is running, the function is invoked. The return value of the function determines what happens next.

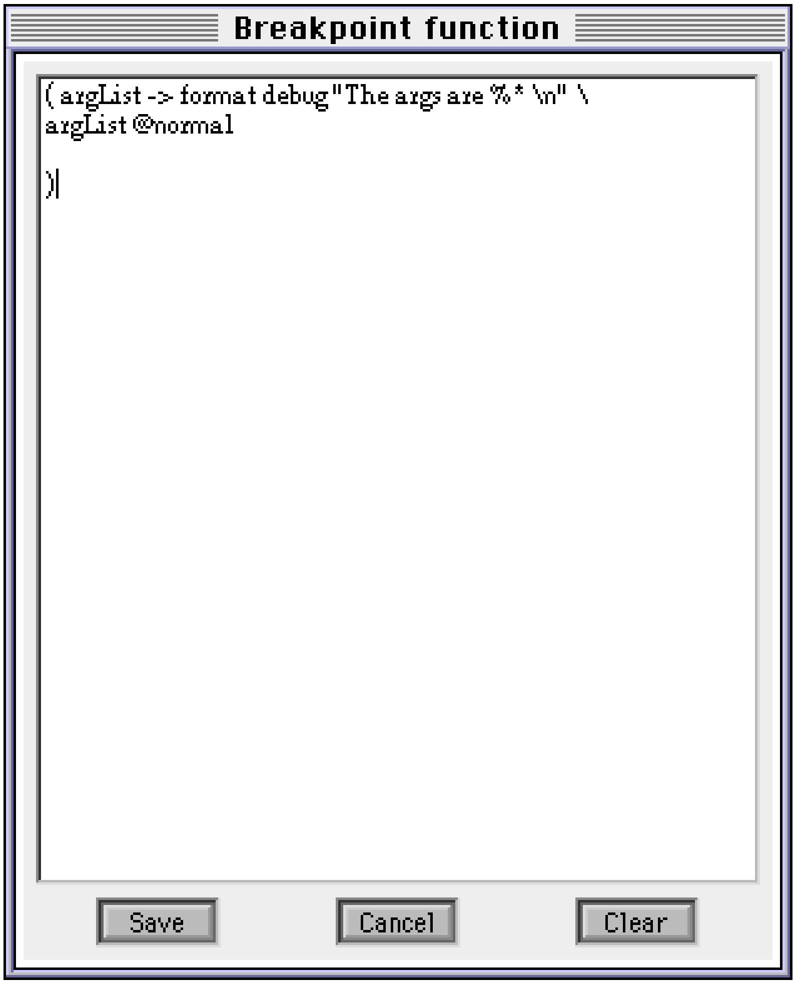
If the return value is `false`, the breakpoint is enacted, and the debugger waits at that expression.

If the return value is anything other than `false`, the program continues running as if there was no breakpoint. In this case, if the breakpoint is encountered while stepping through code in the debugger, the debugger executes the expression and waits. If the breakpoint is encountered while the debugger is not active, the debugger is not invoked.

Figure 9-7 shows a breakpoint function defined in the Edit Breakpoint window.

To remove a breakpoint function from a breakpoint, display the function in the Breakpoint Window, delete all the text for the function, and close the window.

If you remove a breakpoint, its associated breakpoint function is also deleted if there is one.



```
( argList -> format debug "The args are %* \n" \
argList @normal

)
```

Figure 9-7: The Edit Breakpoint window showing a breakpoint function

Using Watchpoints

You can use the debugger to monitor the values of global variables or expressions. These are known as watchpoints.

You can add and remove watchpoints in the Watchpoints window. To open the Watchpoints window, from the **Debugger** menu select **Show** then **Watchpoints**.

To add a watchpoint, click the **Add** button. A watchpoint can be a global variable or an expression. The following are all valid watchpoints :

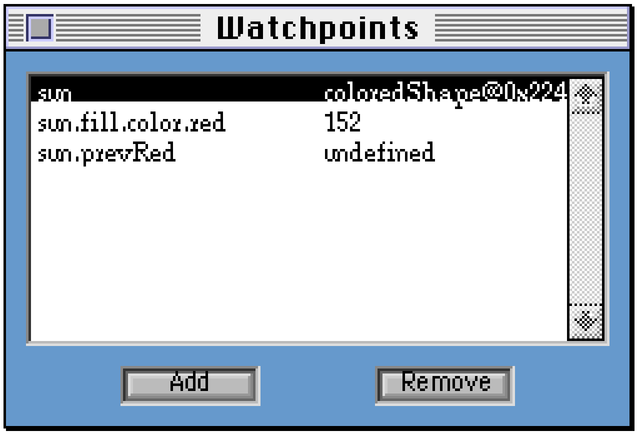
```
x
currentColor
recipe1.ingredients
(person1.hoursWorked * person1.hourlyWage)
b * foogle(c)
```

The Watchpoints window updates the values for all watchpoints periodically. You can determine the time between updates by specifying a value for the **Seconds between updates** field in the Preferences dialog box. A value of 0 indicates that no updates will occur until the debugger is invoked, or a breakpoint is reached. To open the Preferences dialog box, select the **Preferences** command from the Tools menu.

The Watchpoints window updates the values for all watchpoints whenever the debugger is invoked or a breakpoint is reached.

You can cause the Watchpoints window to update the values of its watchpoints at any time by choosing the **Refresh** command in the **Debugger** menu.

Figure 9-8 illustrates a Watchpoints window which is monitoring several global variables and expressions.



sun	coloredShape@0x224
sun.fill.color.red	152
sun.prevRed	undefined

Figure 9-8: The Watchpoints Window

Investigating Exceptions

When ScriptX code is running in the ScriptX development environment, and an error occurs, an exception is reported. You can use the `guard` construct discussed in the *ScriptX Language Guide* to trap exceptions and take appropriate actions. The *ScriptX Class Reference* lists all the exceptions that are pre-defined in the ScriptX environment.

Often you can gain useful information about an error from the message that is printed when an exception is reported. Sometimes though the message by itself is not enough to help you figure out what went wrong. In such cases, you can use the debugger to explore the actions that lead up to the exception.

When the debugger is active, the standard exception reporting mechanism is replaced by a call that invokes the debugger.

Note – When the debugger is active, it intervenes in the case of all exceptions, even if the exception occurs inside a `guard` clause. To resume the program flow and allow the `guard` clause to handle the exception, press the **Go** button in the debugger.

Use the debugger to examine the arguments and variables that were used and bound by the frames leading up to the exception. The Call Stack Menu shows the functions and methods that were called prior to the function or method that caused the error. You can select any call in the Call Stack Menu to cause the Frame Variables panel to display the arguments it was called with and to see the local values that werepushed onto the stack during its execution.

You can set breakpoints on specific ScriptX functions or methods so that next time the code runs, it will automatically stop running at the breakpoints, thus allowing you to step through the remaining code one expression at a step.

To set a breakpoint on any function or method written in ScriptX, ensure that the **Debugger** menu is available, and choose the **Breakpoints** option of the **Show** submenu. In the dialog box that comes up, enter the function or method to break on, as discussed in "Setting Breakpoints" on page 103.

When an exception occurs, you can use the debugger to identify the function or method that causes the error, set a breakpoint on that call, and run the code again. When the breakpoint is reached, you can step through the code to see what changes occur right up to the point that the exception is reported.

When you have finished investigating and setting breakpoints, you can leave the debugger by pressing the **Go** button.

When an exception occurs, you cannot continue stepping through the ScriptX code, since if you could, it would take you into debugger code, which would not be helpful to you in debugging your own code.

Accessing Source Code for Methods and Functions

When you load script files in to the ScriptX development environment every function or method in the source file becomes a `ByteCodeMethod` object. The same is true when you define functions and methods directly in the Listener.

Each `ByteCodeMethod` object has an instance variable called `debugInfo`, whose value is a `DebugInfo` object. Each `DebugInfo` object has a `source` instance variable, whose value is a string of the source code for the function or method that the `ByteCodeMethod` object implements.

If you define functions or methods in the Listener, or you load files either by using the **Open** menu command, or by using the `fileIn` function without specifying the `debugInfo` keyword argument, the source code for the functions and methods is put into `debugInfo` objects. If you do not want to keep the source code, load the file by using the `fileIn` function with the optional keyword argument `debugInfo` set to `false`. (The default is `true`.)

For example:

```
fileIn theScriptDir name:"myfile.sx"
```

loads the file `myfile.sx`, and puts the source code for the functions and methods defined in the file into `debugInfo` objects.

```
fileIn theScriptDir name:"myfile.sx" debugInfo:false
```

loads the file `myfile.sx` without keeping the source code for the functions and methods defined in the file.

Note – If you load a file without keeping the debugging information (that is, the `debugInfo` keyword is `false`), you will not be able to view source code for the functions and methods defined in that file in the debugger.

To see the source code for a function or method, get the value of the `source` instance variable of the `debugInfo` object for the `ByteCodeMethod` object that represents the function or method.

For example, suppose `fnA` is a user-defined function. If you type the following in the Listener:

```
fnA
```

The return value will be something like:

```
#<ByteCodeMethod Scratch:fnA of 3 arguments>
```

If you type the following in the Listener:

```
fnA.debugInfo
```

The return value will be something like:

```
DebugInfo@0x1d5e8f8
```

If you type the following in the Listener:

```
fnA.debugInfo.source
```

The return value will be the a string showing the source code, for example:

```
"function fnD a b c ->(
   w := new window
   w.width := 100; w.height := 100
   tp := new TextPresenter\
      boundary:(new rect x2:100 y2:100) \
      fill:(new brush color:yellowcolor) \
      target:((a as string) + " " +
         (b as string) + " " + (c as string))
   prepend w tp
   show w)"
```

Getting the ByteCodeMethod Object for a Method

To access the source code for methods, you first need to use the method `methodBinding` to get a `byteCodeMethod` object. For example:

```
(methodBinding coloredThing whiten).debuginfo.source
```

returns:

```
"
method whiten self {class ColoredThing} ->
(
   local thisColor := self.fill.color
   thisColor.red := 255
   thisColor.blue := 255
   thisColor.green := 255
   notifyChanged self true

)
"
```

DebugInfo Objects in Title Containers

All `ByteCodeMethod` objects saved to a title container save their `debugInfo` objects to the title container too. Although the debugging information can be useful for debugging during the development process, you might want to create the final version of your title container without saving the `debugInfo` objects. This prevents end users from being able to view the source code for your functions. Also, `debugInfo` objects take up disk space, so they increase the size of the title container file.

To create a version of the title container that does not contain `debugInfo` objects, load the files that make the title container file by using the `fileIn` command with the `debugInfo` keyword argument set to `false`.

You can also call the method `removeDebugInfo` on a `ByteCodeMethod` object to remove its debugging information. For example:

```
for i in (allInstances ByteCodeMethod) do
   removeDebugInfo i
```

Reference

This section provides a reference to the Menus, Windows, and Buttons used by the debugger.

The Debugger Menu

The **Debugger** menu provides commands to help you use the debugger. The **Debugger** menu should appear when the debugger is active.

You can make the **Debugger** menu appear at any time while the Debugger is loaded by selecting the **Debugger** command in the Tools menu.

Show

Lets you select a window to show. The windows list includes all existing debugger windows, including open ones, the Breakpoints window and the Watchpoints window.

If the selected window is not open, this command opens it. If the selected window is already open, this command brings it to the front.

Remove

Lets you select a thread from which to remove a debugger. Only threads with associated debuggers are listed.

Interrupt

Lets you select a thread to interrupt. The debugger opens to show the code that is currently running in that thread. This can be useful, for example, when the main thread is taking a long time to do something, and you want to interrupt it to see what it is doing.

Refresh

Updates the values of the watchpoints being monitored in the Watchpoints window.

Compile

Opens a dialog box that lets you choose a file to load and compile. Methods in the file will be loaded with their source code so the source can be viewed in the debugger.

Resume Callback

Resumes a callback that was cancelled by the debugger.

If a breakpoint or exception is encountered while running a periodic callback, the debugger automatically cancels the callback, so that the breakpoint or problem will not be recursively incurred. When you are ready to continue executing your code, you can use the **Resume Callback** command to reschedule the callback.

You would only want to reschedule the callback if you took away the break point inside it, or fixed and recompiled the code that was causing the exception.

Tools Menu

The **Tools** menu includes commands that are relevant to all tools. It also allows you to select which additional menu to display.

About Debugger

Displays information about the debugger.

Preferences

Allows you to set preferences for the debugger.

You can specify whether or not the debugger should open automatically when an exception occurs. (The default is for it to be automatically invoked.) You can also specify the time between updating for watchpoints. You can specify whether or not the debugger window should close when the **Go** button is pressed.

The preferences window also allows you to specify the font and font size used in the debugger.

If you select the **OK** button in the Preferences window, the preferences are set but not saved, which means that next time you start ScriptX, the old preferences settings will be used.

If you select the **Save** button, the preference settings are set and saved. The system writes out the settings to a file called `debugger.ini` in a directory called `toolpref` in the ScriptX startup directory. If the file `debugger.ini` already exists, the system overwrites it with the new settings. If the file does not already exist, the system creates it, and if necessary creates the `toolpref` directory. The next time you start ScriptX, the system looks in the `debugger.ini` file for the preference settings.

Quit

Quits from the current tool. For the debugger, it closes the debugger, removes the **Debugger** menu if it is displayed, and removes the **Debugger** command from the Tools menu.

Browsers

This menu only appears if the browsers are loaded. Lets you choose a browser to be the current tool.

Debugger

Lets you choose the debugger as the current tool. If the debugger is already the current tool, the **Debugger** command is marked with a tick.

The Debugger Window

The debugger window lets you display and investigate ScriptX source code. The debugger opens when a breakpoint is encountered, the `break` function is called explicitly, or when an exception is encountered (depending on the **New Debugger on Exception** flag.) You can also open the debugger by selecting `Interrupt` from the Debugger menu, or by entering `break ()` in the Listener.

If the **New Debugger on Exception** flag is set to true in the Preferences window, a new debugger will automatically open when an exception occurs in a thread, if there is no debugger for that thread already.

You can have one debugger window per thread.

This section lists the buttons in the debugger.

Go

This button causes the current program to run as it would in the absence of the debugger. If the debugger is active, the program will effectively leave the debugger and continue running.

If an exception is encountered, the behavior in the absence of the debugger is for the Listener to print a warning message and halt the thread that reported the exception. Thus if you press the **Go** button after an exception has occurred, the Listener will print the warning message for the exception and halt the current thread.

The keyboard accelerator for the **Go** button is F5.

Step Into

Executes the next ScriptX expression then halts. If the expression is a call to a function or method, then the debugger steps into that function or method and you can step through the code for it.

The keyboard accelerator for the **Step Into** button is F8.

Step Over

Executes the next expression to completion. If the expression is a call to a function or method, then the function or method is executed to completion.

Note that you cannot "step over" statements. Each expression in a statement is executed as a single step.

The keyboard accelerator for the **Step Over** button is F9.

Out

This button executes the remaining expressions within the current function or method to completion, and halts on the calling frame.

The keyboard accelerator for the **Step Over** button is F6.

The Watchpoints Window

The Watchpoints window allows you to add, remove and view watchpoints. A watchpoint can be any expression.

To open the Watchpoints Window, select the **Show** command in the **Debugger** menu, then select **Watchpoints**.

Add

Allows you to add a new watchpoint to be monitored. In the edit window that appears, enter a watch expression, which can be any legal expression.

Remove

Removes the selected watchpoint.

The Breakpoints Window

The Breakpoints window allows you to add and remove breakpoints, and to define a function to run when the breakpoint is reached while a program is running. When a breakpoint is reached while a program is running, the program halts and the debugger is invoked on the program.

Use the Breakpoints window to specify a breakpoint for a ScriptX function or method. See “Setting Breakpoints” on page 103 for information on other ways to set breakpoints.

To open the Breakpoints window, select the **Show** command from the **Debugger** menu then select **Breakpoints**.

Add

Allows you to add a new breakpoint. In the edit window that appears, enter the name of a function or a method. Specify methods using the format `class^method`. You can set breakpoints for a method on the class the defines the method, but you cannot set breakpoints for a method on a class that inherits the method.

Remove

Removes the selected breakpoint.

Edit Button

Allows you to specify a function to run when the breakpoint is reached.

See “Defining a Function to Run at a Breakpoint” on page 104 for information on defining the function.

CHAPTER

ByteCodeMethod Profiler

10

The ByteCodeMethod profiler provides functions that can help you analyze your ScriptX code for performance problems. The profiler reports information that shows how long methods and functions are taking to run.

How to Install the Profiler

To install the ByteCodeMethod Profiler ensure that the files `prof.sxt` and `prof.lib` are in a folder called `Tools` in the ScriptX startup directory. If these two files are in the `Tools` folder when ScriptX starts, the Profiler loads automatically. If `profs.sxt` is not in the Tools folder when ScriptX starts, you can load it into ScriptX by using the **Open** menu command to open it.

How to Run the Profiler

To run the profiler, you must be in the `Profiler` module. The recommended approach is to create your own module that uses the Profiler module, then work in your own module. You can define your classes, methods, and functions in your own module, then use the profiler to analyze their performance.

The following code shows how to create your own module that uses the Profiler module.

```
module myModule
    uses ScriptX
    uses Profiler
end

-- switch to working in myModule
in myModule
```

To begin profiling, execute:

```
    startProfiler()
```

To stop profiling, execute:

```
     stopProfiler()
```

You can start and stop the profiler more than once to selectively exclude certain portions of your code from profiling.

To see the profiler's report, execute:

```
     profileInfo()
```

To discard the data and prepare for a new profiling session, execute:

```
     resetProfiler()
```

What The Profiler Does

After profiling is enabled, statistics are collected for every call to a ByteCodeMethod object. The time of the initial call is recorded as is the time it returns. The difference between those two times is the total time for that particular call. All calls are tracked in this manner. When you ask for the profiler report, the sum of the times for each ByteCodeMethod object is computed, then any calls to bytecode routines nested inside the original call are subtracted out, yeilding a 'self' time for that object. Note, calls to substrate methods and functions are not subtracted out.

For example, consider the code below:

```
fn fn1 x -> for i := 1 to x do printfn i
fn printfn y -> print y
(startProfiler(); fn1 5; stopProfiler() )
```

If it took 2 seconds for each iteration through the 'for i' loop, and 5 seconds to execute each substrate print statement, the timing of the two functions would look like this:

```
Method     Total  %Total     Self   %Self  Calls
fn1        35     100%       10     28%    1
printfn    25     72%        25     72%    5
```

The total time charged to fn1 would be 35 seconds (2 seconds * 5 + the time charged to printfn), however, the self time would only include the 10 seconds of iteration. The time charged to printfn is the same for both total and self, since it does not call any other bytecode functions (the substrate print function is charged to printfn).

Profiler Output

The profileInfo function prints a report to the Listener that details the time spend in each ByteCodeMethod object. For example, the report might look like the following:

```
Profiled time: 8.817844 sec
Time in thread @theMainThread: 0.017044

Method              Total        %Total     Self       %Self    Calls
table1^initIvs     0.002642      0.029%   0.002642     0.029%     100
Scratch:indexOf    3.503513     39.73%    1.980281    22.45%     1900
Scratch:readUntil  5.598386     63.48%    3.952018    44.81%      500
Scratch:vivName    1.755975     19.91%    1.755975    19.91%     1100
table^append       0.122962      1.394%   0.122962     1.394%     100
table^importCSV    8.817844    100.0%     1.003966    11.38%        1
```

The "Profiled time" is the amount of time used by the profiled threads between the start and stop calls.

A report is generated for each thread for which timing information has been collected. The "method" column lists the bytecode methods that were invoked. The names will either have the form

```
Module:global (functions)
Class^method (methods)
anonymous@addr
```

The "Total" column lists the amount of time elapsed from the time that function was invoked until it returned.

"%Total" is this number as a percentage of the total time.

The "Self" column shows the amount of time actually spent in that bytecode method (and in substrate calls by that bytecode method).

"%Self" is this time as a percentage of total time.

The "Calls" column shows the number of times this bytecode was invoked.

The difference between "Total" and "Self" can be seen by looking at the bottom line of the example. The `importCSV` method took up 100% of the profiled time, however only 11.38% was spent in this method, the rest of the time was spent in methods and functions called by `importCSV`.

The default sorting method for printing the report is by the "Total" column. Use the `sort` keyword option to print the report with other sort keys. The example is sorted by `@name`.

Anonymous Objects

Sometimes the Method column will include anonymous objects, which are shown as: `anonymous@addr`

You can find the object by "objectifying" the address, for example:

```
objectify 30757248
```

Which might return something like:

```
#<ByteCodeMethod anonymous@0x1d55180 of 1 argument>
```

If the anonymous function is a `ByteCodeMethod` object, you can find if it has any debugging information, by getting the value of the `debugInfo` instance variable:

```
(objectify 30757248).debugInfo
```

If it has a `debugInfo` object, you can see the source code for the method or function, if it is available, by getting the value of the `source` instance variable of the `debugInfo` object:

```
(objectify 30757248).debugInfo.source
```

Output Options

The `profileInfo` function supports the following keyword options:

```
sort: { @name | @total | @self | @calls }
stream: <streamObject>
raw: {true | false }
```

The `sort` option specifies the field used to sort the timing information. By default, the `@total` option is used.

The `stream` option allows the output to be redirected to a ScriptX stream. The default value is `debug`, the console stream.

The `raw` option defaults to `false`. If set to `true`, the output is not formatted, but dumped in comma-delimited rows for output to a spreadsheet or other post-processor.

Limitations

The profiler does affect the execution time. The profiler redirects the invocation call on ByteCodeMethod to collect statistics. Each call has the additional overhead of fetching the time, and building a catch frame for each call. However, this overhead appears to be relatively low, in the 2% range.

Example Use of the Profiler

This section discusses one example use of the profiler. It does not delve ito code very much, but is intended to give you an idea of how to assess the profiled information and use it to think about ways to speed up your code.

For an example, consider the results of running the following code:

```
(startProfiler (); flashColor 220 115 170
   stopProfiler (); profileInfo ())
```

The profiled information is:

```
Profiled time: 0.512830 sec
Profile for thread @theMainThread (0.512830)

Method              Total        %Total     Self       %Self   Calls
mod1:flashColor    0.51        100.0%    0.000467     0.091%       1
mod1:FindColoredShapee
                   0.50         97.56%   0.5         97.56%        1
mod1:ColoredThing^flash
                   0.01          0.36%   0.006        1.22%        1
mod1:ColoredThing^restoreColor
                   0.012         0.37%   0.002        0.37%        1
mod1:ColoredThing^blacken
                   0.001635      0.318%  0.0016       0.32%        1
mod1:ColoredThing^whiten
                   0.001318      0.257%  0.0013       0.26%        1
mod1:ColoredThing^saveColors
                   0.001011      0.197%  0.0010       0.2%         1
```

In summary, the function flashColor took just over half a second to run, and half a second was spent in the findColoredShape function. This tells us that speeding up findColoredShape would give us the greatest speed gain.

So we look at the definition for findColoredShape:

```
function findColoredShape red green blue ->
(
   local thisSquare := undefined
   for square in (allInstances coloredShape) do
   (local color := square.fill.color
      if color.red = red and
         color.blue = blue and
         color.green = green
      do thisSquare := square
   )
   thisSquare
)
```

This function looks at every instance of the class `ColoredShape` to see if its color has the desired red, green, and blue values. Even after it has found a match, the function continues testing the rest of the colored shapes.

One way to improve the function would be to redefine `for / in` the loop clause so that it terminates as soon as a match is found. You can achieve this eitherusing `until` or `while` to test for a termination condition, or by rewriting the function to use the `chooseOne` method on a collection. The `chooseOne` method finds one object in the collection that matches the desired criteria.

So we redefine `findColoredShape` as follows:

```
function findColoredShape red green blue ->
(
    local thisSquare := undefined
    chooseOne (allInstances coloredShape) \
    (a b -> local color := a.fill.color
        color.red = red and
        color.blue = blue and
        color.green = green) 1
)
```

Now reset the profiler to clear out the existing information, and run the `flashColor` function again:

```
resetProfiler ()
(startProfiler (); flashColor 220 115 170
    stopProfiler (); profileInfo ())
```

This time the profiled data is as shown below. You see that the execution time has been cut to 0.34 seconds, down from 0.51 seconds. Most of the time is still spent in the `findColoredShape` function, which now takes 0.33 seconds, instead of 0.5 seconds, which is a significant gain. However, `flashColor` still spends most of its time in the `findColoredShape` method, so if we could think of a way of narrowing the search, we could cut the execution time down even more.

```
Profiled time: 0.336595 sec

Profile for thread @theMainThread (0.336595)
Method               Total        %Total     Self       %Self    Calls
mod1:flashColor
                   0.37        100.01     0.00042      0.125%       1
mod1:FindColoredShape
                   0.33         96.56     0.32        96.38%       1
mod1:ColoredThing^flash
                   0.011         3.30     0.0057       1.7%        1
mod1:ColoredThing^blacken
                   0.002         0.58     0.0019       0.58%       1
mod1:ColoredThing^restoreColor
                   0.0016        0.47     0.0016       0.47%       1
mod1:ColoredThing^whiten
                   0.0013        0.39     0.0013       0.39%       1
anonymous@29211952
                   0.0007        0.20     0.0007       0.21%       1
mod1:ColoredThing^saveColors
                   0.00058       0.17     0.00058      0.17%       1
```

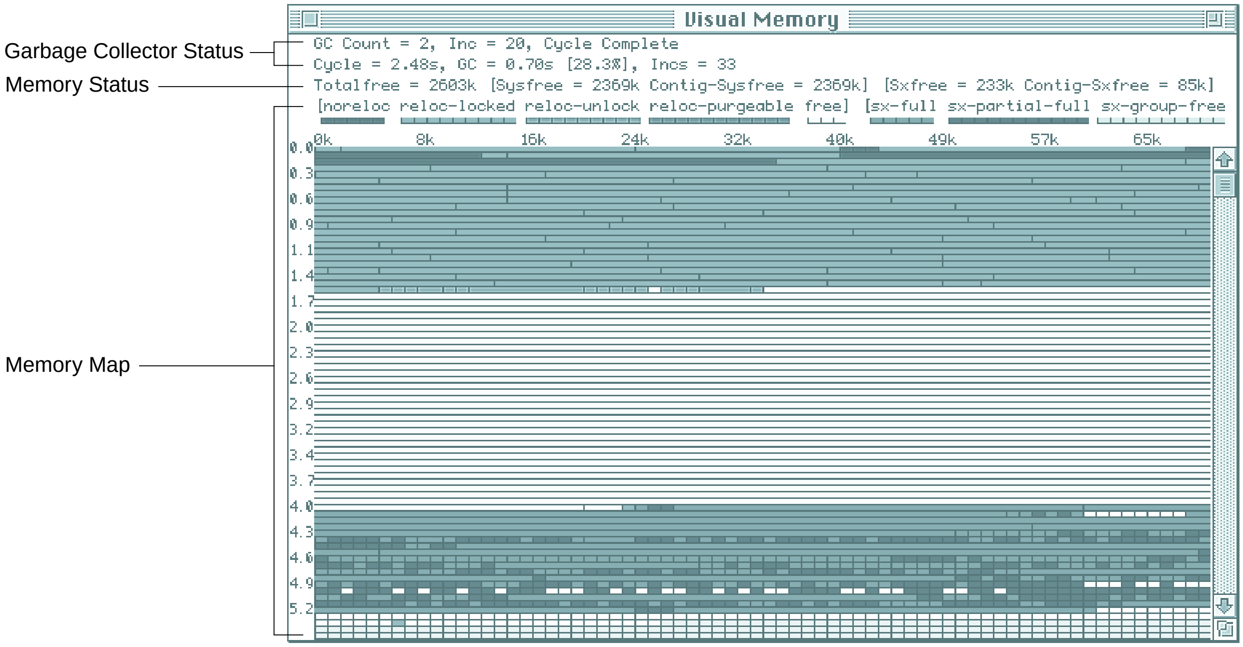
CHAPTER

Visual Memory

11

Visual Memory is a utility for examining memory that is managed by the garbage collector. In the ScriptX executable, choose **Visual Memory** from the File menu in the ScriptX menus to view a visual map of the ScriptX heap.

Figure 11-1 depicts the Macintosh version of Visual Memory, with the upper left portion of the window scrolled into view.



```
GC Count = 2, Inc = 20, Cycle Complete
Cycle = 2.48s, GC = 0.70s [28.3%], Incs = 33
Totalfree = 2603k [Sysfree = 2369k Contig-Sysfree = 2369k] [Sxfree = 233k Contig-Sxfree = 85k]
```

Figure 11-1: Macintosh version of Visual Memory

Platform Differences

The Macintosh and Microsoft Windows versions of Visual Memory are similar in appearance, except that the Macintosh version has a single vertical scroll bar and is resizeable in the horizontal direction.

The Macintosh and Windows versions of Visual Memory differ slightly, reflecting memory management on the underlying operating system. The Macintosh System allocates both pointers, or nonrelocatable objects, and handles, which are relocatable pointers to pointers. Handles can be locked, unlocked or purgeable. Windows allocates only nonrelocatable objects, or pointers. The Windows version of Visual Memory distinguishes between memory that is free, or deallocated, and memory that is unused, that has never been allocated from the Windows global heap. The Macintosh and Windows also differ in internal layout of memory. For more information, consult a reference to the architecture of the underlying operating system.

Running Visual Memory in the Kaleida Media Player

The previous section described how to display Visual Memory in the ScriptX development environment. However, looking at memory can be even more critical in the playback environment where your application must fit in a minimum memory requirement. You can display Visual Memory in the Kaleida Media Player, as follows:

Microsoft Windows: Press the keys Ctrl-Shift-N (in contrast to Ctrl-N in the ScriptX development environment). There is no Visual Memory menu selection for this.

Macintosh: This procedure first requires you to modify the Kaleida Media Player. You must add Visual Memory to the File menu, as follows:

1. Make a copy of the Kaleida Media Player. This procedure adds a new menu item called "Visual Memory" to this copy.
2. Open the Kaleida Media Player in ResEdit.
3. Double-click the MENU resource to open up its window.
4. Double-click resource number 201 (the second of the two File menus).
5. Choose **Create New Item** from the Resource menu.
6. Type "Visual Memory" in the Title field, and type "M" into the Cmd-Key field.

Note – Do not move the menu item up or down or add a separator above it. Leave it at the bottom of the File menu immediately below the Quit option. (This works by the position of the menu item in the menu, not by the menu text.)

7. Save the file and quit ResEdit.

To use Visual Memory, start the Kaleida Media Player and choose "Visual Memory" from the File menu.

Details of Visual Memory

Visual Memory display the status of both memory and the garbage collector.

Garbage Collector Status Lines

All of the values in the top two lines of the Visual Memory window display statistics from the latest garbage collection cycle. The first two items on the top line are described in the following table.

Skipping to the right end of the top line, The current status of the garbage collector is shown at the end of the top line of the Visual Memory window:

```
Cycle Complete
```

which means the garbage collector is not collecting garbage.

When garbage is being collected, a *gray bar flashes on and off*, and the status changes to "Scan Statics", "Scan Gray Set", "Recycle Garbage" and other indicators.

Along the top of the Visual Memory window, just beneath its control bar, ScriptX displays statistics from the latest garbage collection cycle. The following table indicates how to interpret these figures.

Table 11-1: Interpretation of garbage collector status

Item or label	Typical Value	Interpretation
`GC Count`	`23`	number of cycles the garbage collector has completed since startup
`Inc`	`20`	length of one increment, in milliseconds
`Cycle`	`18.01s`	length of the most recent garbage collection cycle, in real time
`GC`	`1.27s (7.1%)`	amount of time during the cycle that was actually spent in the garbage collector
`Incs`	`61`	number of increments to complete the most recent cycle
`max inc`	`21.26`	length of maximum increment in the most recent cycle
`wb`	`191.5ms (1.1%)`	amount of time spent in write barrier in the most recent cycle

Memory Status Lines

The third line in the Visual Memory window indicates the size of various regions of memory:

Totalfree – Total amount of memory, combining both system and ScriptX heaps

Sysfree – Total free memory in the system heap

Contig-Sysfree – Continguous free memory in the system heap

Sxfree – Total free memory in the ScriptX heap

Contig-Sxfree – Contiguous free memory in the ScriptX heap

Multicolored Memory Map

In Figure 11-1, each small rectangle represents a page in memory, 512 bytes in the current version of ScriptX, and contains objects of the same number of bytes. Dynamic memory is at the top; static memory at the bottom.

The colors are defined as follows:

- Red (“sx-full”)– indicates that page is fully allocated and has no room for more objects.
- Blue (“sx-partial-full”) – indicates the page is partially allocated.
- Light Green (“sx-group-free”) – indicates a page that is completely unused.
- White (“sx-any-free”)– represents unused memory that has never been claimed by either static or dynamic memory.
- Turquoise blue (“sx-static”)– static memory.
- Orange (“sx-fakeptr”)– indicates that page is pointed to by a *fake pointer* in the root set. Fake pointers are a natural consequence of conservative collection. Since fake pointers occur in an unused section of memory, we know that they are not real objects. A conservative garbage collector will not always identify every object that is ready to be discarded.
- Yellow (“free”) –

- Brown (“noreloc”) –
- Purple (“reloc-locked”) –
- Gray (“reloc-unlocked”) –
- Dark green (“reloc-purgeable”) –

Refer to the flowchart, . When a cell toggles between blue and red, that means an object is being allocated (red) and freed (blue). As described below, click on that cell when it’s red and again when it’s blue to get lists of what it contains. Compare the lists to find out which object is actually being allocated and freed.

Use `findParents` when the memory state is static; use the previous technique when the state is changing.

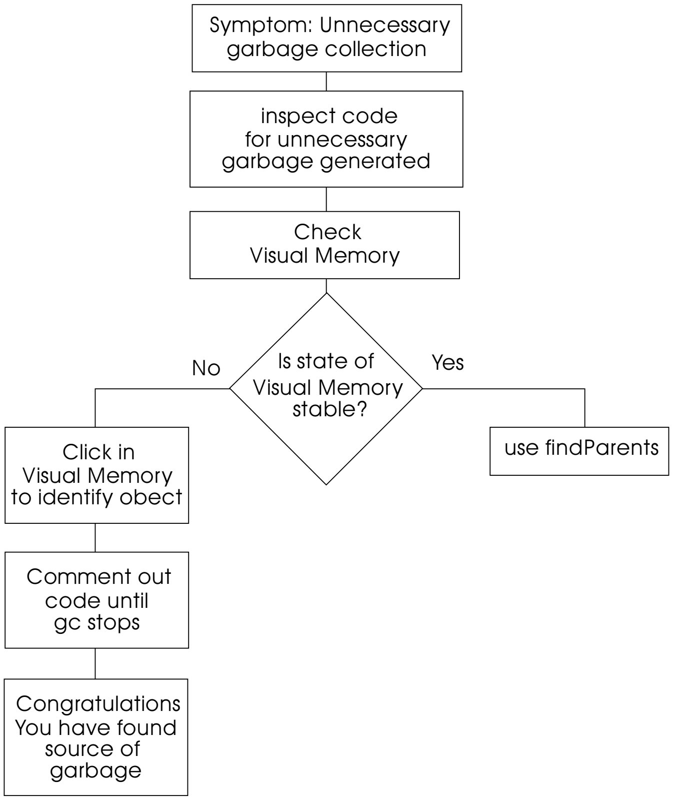

Symptom: Unnecessary garbage collection
inspect code for unnecessary garbage generated
Check Visual Memory
Is state of Visual Memory stable?
No
Yes
Click in Visual Memory to identify obect
use findParents
Comment out code until gc stops
Congratulations You have found source of garbage

Figure 11-2: Flowchart for troubleshooting unnecessary garbage collection

Examining Pages in Detail

To examine the contents of a page of memory:

- Click on the cell that represents it in the Visual Memory display.

ScriptX prints to the Listener window a list of objects that are allocated on that page. ScriptX stores objects of the same size, though not necessarily of the same type, together on a page. Objects that occupy more than one page are stored in contiguous pages.

“SXObjects” indicates chunks of memory for no particular class.

CHAPTER

Tool Framework

12

Important – This document describes a preliminary version of the ScriptX Tool Framework. Both the framework and this document are provided to help developers implement simple tools for internal use and begin development of ScriptX-based tools. Key parts of the Tool Framework described here are likely to change significantly in a subsequent ScriptX release.

Note-When using the ScriptX Tool Framework, you will need to rename the "TOOLSX" directory to "TOOLS" before launching ScriptX in order to have access to the Tool Framework and to use tools built with it (such as the Debugger and Browser).

Overview

This section describes the approach to tool development embodied in the current ScriptX Tool Framework.

ScriptX Tool Support

The design of the ScriptX Tool Framework reflects the need to support a variety of types of tools. The following discussion first looks at the types of tools that will be required to develop multimedia with ScriptX, then reveal how the ScriptX Tool Framework supports those types of tools.

Types of Authoring Tools

We can imagine five categories of tools that can be used in the creation of a ScriptX title: language tools, metaphor tools, component tools, media editors, and title tools. Of these, this release of the Tool Framework includes only language tools. (The category can be specified by the `category` keyword in `ToolContainer`.)

Language tools, such as a debugger, browser, and compiler, are low-level tools that any ScriptX programmer needs to be productive. Kaleida will provide third-parties with language tools.

Metaphor tools describe paradigms for user interaction and allow the user to author using high level tools. Typically, a metaphor tool consists of a set of classes that describe the paradigm, a metaphor editor, and several component editors to manipulate the paradigm's classes. Existing authoring metaphors include timelines and cards and stacks.

Component tools are designed to edit one kind of object or behavior. These tools can stand alone or be one part of a more integrated environment. The developer of a metaphor tool might provide a few component tools to edit classes specific to the authoring metaphor.

Media editors create and manipulate sound, image, and movie data. Presently, ScriptX supports media editors through the importers, which accept most common media formats on both Macintosh and Windows.

Title Tools exist as part of a title and not in the ScriptX authoring environment. They allow end users of a title to change the title's contents. For example, a photo album title might have simple tools for arranging pictures and for adding captions. Authors of such tools should ignore this document and instead refer to the *ScriptX Class Reference*.

Tool Framework

The Tool Framework supplies a platform on top of which tools can be built. In addition to defining a basic protocol for a tool, the framework gives tools access to several host-operating system features, like the menu bar, and provides a substrate so that multiple tools can run simultaneously and give the user the impression that there is just one big tool, the ScriptX environment.

This framework is independent of both metaphors and protocols for titles. The tools that live in this environment are expected to provide these behaviors. Other tools that wish to take advantage of a specific metaphor or special class should implement the protocols provided by those objects.

Many of the features in the framework simplify tasks that most tools will need to perform, such as window management and trapping mouse events. Each tool could implement these features independently, but the framework makes creating tools even easier than it already would be in ScriptX.

Tool Protocols

To effectively send messages between tools and titles, a set of protocols is needed. With the first release of ScriptX, there exists a de facto standard, the ScriptX Core Classes Library, but work should be done in the future to ensure more powerful tools by creating specialized protocol suites.

Presently, tools should use the Core Classes as their protocol. The Core Classes provide a well-rounded set of behaviors and object appearances. Methods and instance variables are guaranteed to have the same names and similar behaviors in all descendants of Core Classes. For example, a tool that knows about the `TwoDPresenter` class can effectively move and resize any object that appears in a `TwoDSpace`, including graphic shapes, bitmaps, user interface objects, and movies.

The model for tool development encourages tool writers to create their own classes and to supply component tools to edit them. Only the specialized tool can edit every aspect of an object's behavior, because the tool has been tailored for the object. But provided that these new classes have some inheritance from the Core Classes, a variety of tools can edit at least parts of them.

Future work should be done by Kaleida and third party developers to establish higher level protocols. This involves identifying major sets of functionality, such as a page layout suite, and creating a layer on top of the Core Classes that allows tools to easily work with objects that conform to that suite. Such standards will evolve naturally as more developers embrace ScriptX.

Writing a Tool in ScriptX

The Tool Framework provides a structure for writing tools. In addition to the classes and services that the framework provides, there are several philosophical elements that you should consider when writing a tool. Following these guidelines will help to ensure a powerful authoring environment for the user.

The first and foremost point about tools is that they are written in the ScriptX language using the Core Classes. The Core Classes provide you with a strong application framework with which to build your tool. Your interface will be written using ScriptX

UI objects, your event handling through controllers and the event system, and your files stored in storage containers. The titles your tool produces shares this foundation. Get to know and love the Core Classes Reference.

When you design your tool, start with a working model of the end result, and work backwards to create an editor for it. Define the classes that your tool will create or edit. Script a simple title using these classes. Then build a tool that will create and edit instances of these objects.

Try to modularize your tool development as much as possible. This effort is key to the interpretability of tools and to the user's ability to pick and choose their preferred tools. For example, if you have a metaphor tool and create separate tools to modify the screen objects, then the user can use your screen objects in another metaphor tool or even use a different editor to change the objects within your tool.

Protocols for good behavior in the ScriptX authoring environment are scarce now, as there aren't really tools available. However, as more tools appear, so will the protocols that they bring with them. As much as possible, use these existing protocols, so that your tool will be the most useful to your users.

By adhering to standard protocols, you can greatly reduce the amount of programming effort required to make a tool. Nearly every metaphor tool contains an asset manager, a layout editor, and a color picker, for example. A well-behaved ScriptX tool does not need to implement these parts, because the user is guaranteed to have some form of each of these tools in his or her authoring environment.

Implementation of the Tool Framework

For a tool to automatically be loaded at startup time, do the following:

1. Make sure there is a directory called `tools` at the same level as the ScriptX executable.
2. Make sure the `tools` directory includes a file named `tlfrmwrk.lib`, which is the tool framework.
3. Put any tools you want automatically loaded into the `tools` directory. The following section describes how the tool startup process works.
4. Start ScriptX.

Tool Startup Process

ScriptX contains code that tries to load in the Tool Framework at startup time. This code's main goal is to load in the framework, but it will also automatically load any tools, titles, and scripts (but not libraries or accessories) that are in the `tools` directory. You can place your own versions of these in the `tools` directory to have them automatically load at startup.

When ScriptX starts up, the following steps take place:

1. This tool-loading mechanism examines only the contents of a directory called `tools` in the same directory as the ScriptX executable. If the `tools` directory isn't found (for example, if it is renamed), ScriptX makes no further attempt to load tools.
2. Then, any ScriptX title containers (`.sxt`) in the `tools` directory are loaded.
3. And then it looks for scripts (`.sx`) in the folder, then loads and compiles them. The order of loading is platform-dependent.

4. If there are directories directly inside of the `tools` directory, files within them are loaded, as well in the same order: first title containers, and then scripts. Note that the contents of any directories below this level are not loaded.

This section describes the features that the Tool Framework includes and shows how a tool might take advantage of them.

Tool Organizer

At the core of the Tool Framework is a global object, `theToolOrganizer`, a global instance of the `ToolOrganizer` class, which oversees much of the functionality of the framework. All tool communication and activation is handled through this object. Furthermore, this object is called from the ScriptX run-time for such actions as menu updating and window switching.

For your tool to fit into this framework, the tool must be registered with `theToolOrganizer`. Registration occurs automatically when a tool is loaded. The instance `theToolOrganizer` provides the methods `register` and `unregister` for explicit registration.

Tool Communication

In order to allow seamless integration of multiple tools, the Tool Framework contains a simple messaging system. This mechanism activates the preferred tool for an object in response to the user's actions. For example, double-clicking a `PushButton` object might bring up a script editor.

This messaging system relies on two methods, `inquire` and `edit`, that the tool must implement, as shown in Figure 12-1. The instance `theToolOrganizer` asks each tool if it can edit an object by calling its `inquire` method; it returns an array of two names describing what it can do. Depending on the return value, `theToolOrganizer` can then call the `edit` method on the tool, passing in the object, enabling the object to be edited by the tool.

Alternatively, to edit an object with the available tools, you can call `inspect` on the object, which is equivalent to `edit theToolOrganizer theObject`.

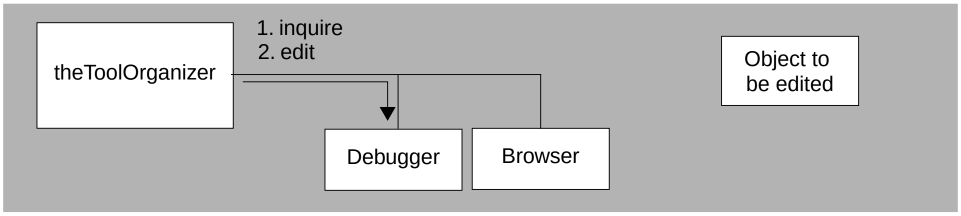


Figure 12-1: The inquire and edit methods.

To implement the default behavior for this context switching, the current tool needs to call the `edit` method in `theToolOrganizer`. This searches through the list of tools and finds a tool that can edit the object. For example, an authoring tool's click handler might look like:

```
global theDbleClickTime := 10
method getClick self { class MyTool } theSpace clickPt theTime ->
(
   local  tmpObj

-- loop through each object in the space.

   for tmpObj in theSpace do
-- check to see if the click is inside any of the
-- objects.

        if PtInBoundary tmpObj clickPt do

-- look for double-click state:  same object and
-- within a certain time difference.

            if ( tmpObj = self.lastClick ) do
                if ( theTime < self.lastClickTime + theDbleClickTime ) do
                (

-- this asks the tool organizer to find an editor for
-- the object and to switch to that tool.  Most likely,
-- we will be switched out at this point.

                    edit theToolOrganizer tmpObj #()
                    return self
                )

-- return something.

    return self
)
```

To support this messaging, each tool must implement two methods: `inquire` and `edit`. The instance `theToolOrganizer` asks each tool if it can edit an object by calling its `inquire` method. The tool can test whatever aspect of the object it wants. Some of the tests a tool might perform are:

```
-- in this example, we will check to see if there is a Brush object
-- that we can edit.  there are several ways we can look for one.

method inquire self { class MyTool } theObject theOptions ->
(
    localivList, tmpName

-- see if there is an instance variable with a given
-- name in the object.  any of the core classes that
-- contain Brush objects call them “fill.”

    ivList := allIvNames theObject
    tmpName := new NameClass name: @fill
    if isMember ivList tmpName do
        return #(@viewAndEdit, @knowPartial )

-- we can check the class directly to see if this object is
-- a Brush object.

    if ( getClass theObject = Brush ) do
        return #(@viewAndEdit, @knowAll )

-- we can also check to see if the object is a subclass
-- of Brush, in which case we can edit at least parts of
-- the object.

    if ( isAKindOf theObject Brush ) do
```

```
        return #(@viewAndEdit, @knowSuperclass )

    return #(@viewAndEdit, @unknown )
)
```

If the tool indicates that it can edit the object by what it returns from the `inquire` method, `theToolOrganizer` can then make the tool current and call its `edit` method on the tool, with the object as the argument. A simple example follows:

```
method edit self { class MyTool } theObject theOptions ->
(

-- bring up our editing window, if we haven't already
-- in our switchIn method.

    show self.editPalette

-- then set the window up to edit this object.  at this
-- point, the user can use the controls in the window
-- to change characteristics of the object.

    updateParams self theObject

-- return something.

    return self
)
```

Tools designed as part of a suite can also explicitly ask other tools to edit objects. For example, if you have a tool for editing custom classes, your main tool can manually load the tool from the object store, make the tool current, and then call its `edit` method.

Tool Messaging

In addition to the support for passing objects to other tools for editing, tools can send messages to inform the others of changes to the system's state. Some examples of the messages are when an object has been changed, so that its owner knows when to redraw it, or when an object has been deleted, so that an editor can close.

The Tool Framework's message system is implemented as two methods in the Tool class. A Tool can call the `sendMessage` method to inform other tools of changes. The Tool should also implement the `getMessage` method to receive messages sent by other tools. Tools do not get messages that they send.

Information included in a message is the object in question and a `NameClass` object that signifies the event. Developers are expected to create suites of events that compatible tools can support. The Tool Framework defines the following messages: `@objectChanged`, `@objectDeleted`, and `@saveTitle`.

Here follows sample implementations of the messaging methods:

```
-- method that will grab incoming messages.  In this
-- method, we will check to see if the object we are
-- editing (we are a Brush tiny tool) has been deleted.

method getMessage self { class MyTool } theObject theMessage ->
(

-- first see if we are even editing an object.
```

```
    if ( self.currentObject = undefined ) do return self

-- because we are storing a reference to our lone
-- edited object, we can just compare it to the incoming
-- object.  otherwise, we could search through collections
-- or even do smarter tests.

    if ( theObject <> self.currentObject ) do return self

-- we fell through the above test, so this means that the
-- given object is our object.  do something in response.

    case theMessage

-- standard message.  someone else edited a part of the
-- object, so let’s refresh to show the changes.

        @objectChanged:
            self.itsWindow.changed := true

-- another standard message.  someone cleared the object,
-- so close the editor.

        @objectDeleted:
            (
                local  tmpWind

                for tmpWind in self.windowList do hide tmpWind
                self.currentObject := undefined
            )

-- this is our own private message that other tools can
-- send to us.  we published this message, we were first, and
-- it made sense, so consequently, it became the standard.

        @useThisBrush:
            (
                self.currentBrush := theObject
                self.itsWindow.changed := true
            )
    end

-- return an object.

    return self
)

-- and here is a code fragment of the message I would
-- send before closing my tool.  the user made changes,
-- so alert others.  they will redraw or otherwise do their
-- dirty deeds.

sendMessage self self.currentObject @objectChanged
for tmpWind in self.windowList do hide tmpWind
```

Easy Event Handling

Many ScriptX tools will need to intercept mouse events in title windows before the title's controllers get at them. For example, if you need to drag around buttons (which are already attached to a controller), how do you do that without activating the buttons? With the Tool Framework, there is a way.

Each `Tool` object has two instance variables called `wantsClicks` and `clickHandler`. The two work together to give your tool instant access to mouse click events in title windows. This mechanism ignores clicks in and instance of `ToolWindow` or its subclasses.

To automatically receive mouse down events, set the `wantsClicks` instance variable of your tool to `true`. If the tool is current, then events are processed immediately. If some other tool is current, then yours will receive events after it is switched in. You don't need to manually toggle the `wantsClicks` setting to handle tool switches.

The method that gets called in response to a click is stored in the `clickHandler` instance variable. It should return `true` if your tool responded to the event and doesn't want the click passed on to the title's controllers. Or you can return `false` and whatever would have happened without your tool present will happen.

```
-- sample clickHandler method.  this just adds
-- objects to our mythical tool's selection list.

method selectClick self { class MyTool } theSurf thePt theTime ->
(
    local      clickedObject

-- use some utility method to run through the
-- presenter hierarchy and get the TwoDPresenter
-- that was clicked on.

    clickedObject := findClicked self theSurf.topPresenter

-- clicked in blank space.  just return.

    if ( clickedObject = undefined ) do return false

-- clicked on some other TwoDPresenter.  see if we
-- care about it.  we can use an already existing
-- method to check.

    editResponse := inquire self clickedObject #()
    if ( editResponse[2] = @unknown ) do return true

-- okay, we really care.  add it to our selection
-- list.

    append self.selectionList clickedObject

-- don't let anything else happen as a result of the click.

    return true
)
```

This event handling is designed so that a tool can switched method in its `clickHandler` based on internal settings. For example, if a tool had modes where clicking on an object selected it, dragged it, or linked it to another object, each behavior could be implemented in a different method. As the user chooses different modes, the tool need only set the `clickHandler` variable to the appropriate method to change the behavior.

Tool Host OS Access

The Tool Framework contains several classes to help tool programmers make tools that behave similarly to applications written on top of the platform's operating system.

Menus

When the Tool Framework has been loaded into ScriptX, a new menu item called **Tools** appears at the end of the menu bar. Every tool that has registered with `theToolOrganizer` is listed in this menu. To select a tool, the user chooses one from the list. The current tool is marked with a check.

ScriptX Menu Bar without Tools Installed

The first two items in this menu, About and Preferences, are standard items that each tool will have. `Tool` subclasses override the corresponding methods: the `about` method should display a dialog that shows the tool's copyright and credits; `preferences` should provide controls to set options for the tool.

Each tool can create and maintain its own menus in the system's menu bar. These menus are listed to the right of the **Tools** menu. When `theToolOrganizer` switches in a tool, it installs its menus. When it switches out a tool, it removes the menus from the menu bar.

ScriptX Menu Bar with Tools Installed

The framework has three classes with which a tool can maintain its menu bar: `ToolMenuBar`, `ToolMenu`, and `ToolMenuItem`. Each tool automatically has an empty menu bar allocated for it. This object is stored in the instance variable `systemMenuBar` inherited from the `ToolContainer` superclass `TitleContainer`.

Anatomy of the ScriptX Menu Bar

A `ToolMenuBar` instance is an array of `ToolMenu` objects. Use standard array manipulation methods to add, remove, and reorder the menus. When you are done changing the menu bar, call the `menuChanged` method of the menu bar.

A `ToolMenu` object is an array of `ToolMenuItems`. As with `ToolMenuBar`, use array methods to add or remove items from a menu. Each `ToolMenu` object has an instance variable called `menuName` that is displayed in the menu bar. If you change the contents of a menu or its name, call the `menuChanged` method.

The `ToolMenuItem` class defines the behavior of each menu item. All menu updating and execution is performed through callback functions defined by your tool. Each menu item has a single `menuHandler` that `theToolOrganizer` calls. This function is passed a selector which indicates whether it should perform the menu command or return the state of the menu (e.g.. enabled, checked). By default, your tool is the first argument passed to the callback. However, you can specify another object by setting the `owner` instance variable.

The appearance of a ToolMenuItem also depends on the values of its instance variables. itemName contains the text to be displayed in the menu item. Menu items with "-" as their name appear as separating lines. Items can also have keyboard shortcuts, which are specified in the shortCut instance variable.

The following code shows the basic menu handling for a typical tool:

```
-- method to set up the menu bar for this tool.  add
-- the menus and menu items.

method makeMenus self { class MyTool } ->
(

-- create the pasta menus.  this allows the
-- user to choose one of three pastas.  the
-- current one is checked.

   self.firstMenu := new ToolMenu name: “Pasta”
   append self.firstMenu ( new ToolMenuItem \
      name: “Linguine” menuFunc: pastaFunc \
      shortCut: “l” )
   append self.firstMenu ( new ToolMenuItem \
      name: “Fettuccine” menuFunc: pastaFunc \
      shortCut: “f” )
   append self.firstMenu ( new ToolMenuItem \
      name: “Cappellini” menuFunc: pastaFunc \
      shortCut: “i” )
   append self.menuBar self.firstMenu

-- make the second menu, for sauces.

   self.secondMenu := new ToolMenu name: “Sauces”
   append self.secondMenu ( new ToolMenuItem \
      name: “Pesto” menuFunc: sauceFunc )

-- put a divider line in to separate the healthy
-- sauces from the really unhealthy ones.

   append self.secondMenu ( new ToolMenuItem \
      name: “-” )

-- add more items, then put the menu in the
-- menu bar.

   append self.secondMenu ( new ToolMenuItem \
      name: “Alfredo” menuFunc: sauceFunc )
   append self.secondMenu ( new ToolMenuItem \
      name: “Carbonara” menuFunc: sauceFunc )
   append self.menuBar self.secondMenu

-- return this object.

   return self
)

-- the callback function for the items in the
-- pasta menu.  they all use the same callback.
-- this gets called both for updating AND performing.
```

```
method pastaFunc self { class MyTool } mItem options ->
(

-- options specifies what the tool organizer wants us
-- to do.  we can return one of four values:  @enabled,
-- @dimmed, @enabledChecked, or @dimmedChecked.

    if ( options = @update ) do
    (

-- to see if this is the current pasta, compare
-- the item number against our previously set up
-- instance variable.  check the current one.

        if ( mItem.itemNum = self.currentPasta ) then
            return @enabledChecked
        else
            return @enabled
    )

-- the second case is to perform the menu item. for
-- this example, we are going to simply update
-- our pasta selection.  in other tools, we would
-- open a dialog box, run a title, or do some other
-- spiffy thing.  the next time the menus are updated,
-- this item will be checked.

    self.currentPasta := mItem.itemNum

-- return this object.

    return self.
)
```

The DragRegion Class

The class `DragRegion` provides tools with a means of exchanging information between tool and title windows. Using this class, the user can drag and drop objects between windows. The `DragRegion` class only provides the graphical interface for this action—it includes no semantics for copying or placing the actual objects.

To use the `DragRegion` class, first create a new instance with a rectangle as the argument to its `boundingBox` keyword. In response to a mouse down event in that rectangle, your controller should call the `dragMe` method. After the user releases the mouse button, the instance variables of the `DragRegion` object contain information about where the region was dropped.

DragRegion Interface

To the user, a `DragRegion` appears as a rectangle with a gray boundary and no fill. He or she can drag this shape anywhere on the screen. If the dropped location does not fall within a window, then the drag is considered canceled.

```
-- method for dragging a TwoDPresenter object
-- between windows.  call this from your event
-- handling code, after a mouse down.
```

```
method dragShape self { class MyTool } theShape ->
(
   local  tmpRgn, didDrag, tmpRect

-- make the region object and do the drag.

   tmpRect := theShape.bounday
   tmpRgn := new DragRegion boundingBox: tmpRect
   didDrag := dragMe tmpRgn

-- return if the user dropped the shape into
-- a non-scriptx window or the desktop.

   if ( didDrag = false ) do return self

-- test to see if it is a title window.  there
-- are cases where it’s okay to drop into a tool
-- window - that’s up to you to decide.

   if ( isToolWindow theToolOrganizer tmpRgn.dropSurf )
      do return self

-- we might want to place this object into that
-- surface.

   theShape.x := tmpRgn.dropX
   theShape.y := tmpRgn.dropY
   append tmpRgn.dropSurf.topPresenter theShape

-- return an object

   return self
)
```

To drag objects within title windows, use the `DragController` Core Class and the `Dragger` mixin protocol. This ensures compatibility with future versions of ScriptX players.

Future Directions

The following possible areas for expansion of the ScriptX Tools Framework are based on initial user response and fairly obvious shortcomings of the framework.

To behave like normal authoring tools, a ScriptX tool should have access to some of the system menus, in addition to its own private menus. Items of interest to tools are **New**, **Save**, **Open**, **Cut**, **Copy**, **Paste**, and others. To implement this, the menu dispatching would need to be modified to send these commands to the current tool, rather than to ScriptX. In itself, this is not much work. However, adding `Clipboard` support for images and movies (commonly pasted data types), requires some crafty stream and importer tweaking.

In-place media editing promises to be a powerful interface for designing content. The most expedient way to support this ability seems to be through one of the compound document architectures, such as OpenDoc or OLE. However, technical, schedule, and general paradigm constraints make such solutions infeasible for the first release of the framework. When these technologies are integrated into ScriptX, tools that support the Tool Framework will behave similarly to these compound tools.

As more developers begin to develop native ScriptX tools, tool management for the user will get difficult. For example, the **Tools** menu might list dozens of tools, which the user may or may not always need. A solution to help this problem would be to add types to tools, so that they may be categorized for easier access. Some example categories in the `ToolOrganizer` class might be: `@metaphorTool`, `@languageTool`, `@privateTool`, and `@componentTool`. With this categorization in place, the framework could hide inappropriate tools and organize the menu more intelligently.

CHAPTER

Tool Framework API

13

This chapter lists global variables and classes defined in the Tool Framework.

Global Functions

inspect (global function)

`inspect` *theObject theOptions* ⇨ `Boolean`

theObject	Object to edit
theOptions	`Collection` of `NameClass` objects

Equivalent to `edit` *self theObject*.

Sends a request to the tools to edit the given object. In order to find an editor, the `ToolOrganizer` calls each tool's `inquire` method. Using the results of this search, the `ToolOrganizer` picks the most likely candidate, and calls that tool's `edit` method with the given object as the argument. `theOptions` is a `Collection` of `NameClass` objects that represents the context of the edit. If an editor or viewer cannot be found for *theObject*, then edit returns `false`, otherwise it returns `true`.

For more information, see the `edit` method in the `ToolOrganizer` class.

Global Variables

theToolOrganizer (global variable)

`theToolOrganizer` (read-only) `ToolOrganizer`

An instance of the class `ToolOrganizer` which is created at startup when the Tool Framework is loaded. As an `Array`, an instance of `ToolOrganizer` contains a list of all registered tools. If this value is `undefined`, then the framework has not been successfully loaded.

All tools use `theToolOrganizer` as a message center for coordinating with other tools. This object also acts as a gateway between system services (menus and windows) and the tools.

This global variable is actually called `gToolOrganizer` in the current release.

ToolOrganizer

Class type: Tool class (concrete)
Resides in: ScriptX executable but not KMP
Inherits From: `Array`
Component: Tool Framework

The Tool Framework creates a global instance of this class, `theToolOrganizer`, which all tools use as a message center for coordinating with other tools. This object also acts as a gateway between system services (menus and windows) and the tools. You should never create your own instance of this class.

As an `Array`, an instance of `ToolOrganizer` contains a list of all registered tools.

Instance Variables

currentTool

self`.currentTool` (read-only) *(object)*

This instance variable contains a reference to the `ToolContainer` instance of the currently selected tool. Use the `toolSwitch` method to change this value. By default, the value is `undefined`.

categories

self`.categories` (read-only) `Array`

This instance variable contains an array of the categories of the tools that have been registered with the `theToolOrganizer`.

clickInstalled

self`.clickInstalled` (read-write) `Boolean`

Indicates whether the `theToolOrganizer`'s click handling is on (`true`) or off (`false`).

itsToolMenu

self`.itsToolMenu` (read-only) `Array`

Contains an `Array` of the menu items in the **Tools** menu.

Instance Methods

edit

`edit` *self theObject theOptions* ⇨ `Boolean`

self	`theToolOrganizer` global instance
theObject	Object to edit
theOptions	`Collection` of `NameClass` objects

Sends a request to the tools to edit the given object. In order to find an editor, `theToolOrganizer` calls each tool's `inquire` method. Using the results of this search, `theToolOrganizer` picks the tool that knows the most about the object (as described below), and calls that tool's `edit` method with the given object as the argument. The argument *theOptions* is a `Collection` of `NameClass` objects that represents the context of the edit. If an editor or viewer cannot be found for *theObject*, then edit returns `false`, otherwise it returns `true`.

The `inquire` call can return one of these responses: "I know all about the object" (`@knowAll`), "I know the superclass of the object" (`@knowSuperclass`), "I know something about the object" (`@knowPartial`), "I know about objects in general." (`@knowByDefault`), or "I know nothing about the object" (`@unknown`). These responses are listed in preference order, so that the tool that knows the most will be chosen by the tool organizer to edit the object. If two or more tools tie for knowing the most, the first tool that responded will get selected (which is non-deterministic).

findEditor

`findEditor` *self theObject theOptions* ⇨ `Tool`

self	`theToolOrganizer` global instance
theObject	the object to be edited
theOptions	`Collection` instance containing options

Looks through both the tool list and the component list to find an appropriate editor for *theObject*. This routine calls each registered tool's `inquire` method to find the best editor. *theOptions* is a `Collection` of `NameClass` that represents the context of the edit. Editors are ranked as follows, in order of suitability:

```
@knowAll
@knowSuperclass
@knowPartial
@knowByDefault
@unknown
```

If no editor can be found, then `findEditor` returns undefined. See the `inquire` method for `Tool` for more details about this subject.

isToolWindow

`isToolWindow` *self theSurface* ⇨ ??

self	`theToolOrganizer` global instance
theSurface	`DisplaySurface` instance

Returns `true` if the given `DisplaySurface` is being managed by a `ToolContainer`.

register

`register` *self theTool* ⇨ ??

self	`theToolOrganizer` global instance
theTool	`Tool` object to register

Adds the given tool to the list of tools the `theToolOrganizer` knows about. When a tool is registered, its name is appended to the **Tools** menu. If this is the first tool to be registered, then it is switched in automatically.

toolSwitch

`toolSwitch` *self theTool* ⇨ ??

self	`theToolOrganizer` global instance
theTool	`Tool` object to switch to

Makes the given tool the current tool. This tool's menus are placed in the menu bar the tool's `switchIn` method is called, and the previous tool's `switchOut` method is called. If the tool's `wantsClicks` instance variable is `true`, the click handling mechanism is installed.

This method is called by the `theToolOrganizer` in three cases. When the user chooses a tool from the **Tools** menu, the selected tool is switched in through this method. If the user brings a tool's window to the front, the tool is switched in. Finally, calling `edit` switches to the tool that will edit the object.

unregister

`unregister` *self theTool* ⇨ ??

self	`theToolOrganizer` global variable
theTool	`Tool` object to remove

Removes the given tool from the `theToolOrganizer`'s list of tools. A removed tool is no longer listed in the **Tools** menu, nor is it polled for editing objects or for general messages.

version

`version` *self* ⇨ `String`

Returns a `String` representing the version of the Tool Framework.

ToolContainer

Class Type:	Tool class (concrete)
Resides in:	ScriptX executable but not KMP
Inherits From:	`TitleContainer`
Component:	Tool Framework

Creating and Initializing a New Instance

To create a new instance of `ToolContainer`, call `new` on it and supply at least the filename:

```
myTool := new ToolContainer path:"browser.sxt"
```

The variable `myTool` contains the initialized tool container; this expression also creates a file named `browser.sxt` in the ScriptX startup directory (by default). The `new` method uses the keywords defined in `init`.

init

`init` *self* [`dir:` *dirRep*] `path:`*collectionOrString* [`name:`*string*]
[`user:`*libraryContainer*] [`targetCollection:` *collection*]
[`category:`*string*] ⇨ *self*

self	`ToolContainer` instance
`dir:`	`DirRep` instance
`path:`	`String` or collection of strings representing the path
`name:`	`String` representing the filename
`user:`	`LibraryContainer` object
`targetCollection:`	`Collection` object
`category:`	`String` object such as “components” or “tools”

Initializes the instance of the `ToolContainer` and sets the `clickHandler` flag to `false`. You can create a new category by supplying a new string with the `category` keyword.

Instance Variables

clickHandler

self`.clickHandler` (read-write) *(function)*

Specifies a method to be called in response to a mouse click in a title window. This method is called by the `theToolOrganizer` if this is the current tool and its `wantsClicks` variable is `true`. The tool can change this method at any time. The method syntax should take the form:

`aHandler` *theTool theSurface thePt theTime*

theSurface is the `DisplaySurface` in which the mouse was clicked. *thePt* is a `Point` object in local coordinates of *theSurface*. *theTime* is the time on the system clock when the click happened. The `clickHandler` method should have a `Boolean` return value. The function should return `false` if the tool doesn’t want the click, or wants to let the title respond to the click. It should return `true` if the tool took the event and doesn’t want it passed on to the title.

selectionList

self.`selectionList`	(read-write)	`Collectionn`

Place holder instance variable to enable `Tool` subclasses to create and maintain a selection list. The `Tool` class leaves this value undefined.

wantsClicks

self.`wantsClicks`	(read-write)	`Boolean`

A flag that indicates whether or not this tool wants the `ToolOrganizer` to call its `clickHandler` on mouse down events in non-ToolWindows. By default, this value is `false`.

Instance Methods

edit

`edit` *self theObject theOptions* ⇨ ??

self	`ToolContainer` instance
theObject	Object to edit
theOptions	Collection of `NameClass` objects

Requests that the `ToolContainer` edit the given object. Just previous to making this call, the caller will have switched in the `ToolContainer`. Generally, this method is called by the `theToolOrganizer` after it determines that this tool is the right editor for *theObject*.

The argument *theOptions* is a `Collection` of `NameClass` objects which describe the context or parameters for the edit. Currently, there are no predefined options.

getMessage

`getMessage` *self theObject theMessage* ⇨ ??

self	`ToolContainer` object
theObject	Object this message is about
theMessage	`NameClass` object

This method is called when another tool has used the `sendMessage` method to inform other tools of a change in state. Your `ToolContainer` will not be current when it receives this message. The default method does nothing.

inquire

`inquire` *self theObject theOptions* ⇨`Collection`

self	`ToolContainer` object
theObject	Object about which inquiry is made
theOptions	`Collection` of `NameClass` objects

Examines *theObject* in order to determine whether or not this tool *self* can edit or view *theObject*. A `ToolContainer` can check the class of this object, its superclass, or can check for the presence of a set of methods or protocols.

The argument *theOptions* is a `Collection` of `NameClass` objects which describe the context or parameters for the edit. Currently, there are no predefined options.

The return value is a `Collection` of two `NameClass` objects. The first name describes what the tool can do to the object. Currently, the first name can be one of the following:

```
@view
@edit
@viewAndEdit
```

The second name describes the extent of the knowledge about the object. For the second name, pass one of the following objects:

```
@knowAll
@knowSuperclass
@knowPartial
@knowByDefault
@unknown
```

If `@unknown` is the amount of knowledge, then the value of the first name is ignored, but set it to `@viewAndEdit` for future compatibility.

For example, possible return values to this function are:

⇨ `#(@edit, @knowByDefault)`

⇨ `#(@viewAndEdit, @unknown)`

sendMessage

`sendMessage` *self theObject theMessage* ⇨ ??

self	`ToolContainer` object
theObject	Object that message is about
theMessage	`NameClass` object

Alerts other registered tools about a change to the system. In order to listen to these messages, a `ToolContainer` should implement the `getMessage` method, described below.

The three default messages are:

```
@objectChanged
@objectDeleted
@saveTitle
```

However, more messages will be defined as the set of tools grows. Not every tool need respond to every message.

switchIn

`switchIn` *self* ⇨ ??

Performs necessary actions to make the tool the current tool. By default, a `ToolContainer`'s menu bar is installed and, if `wantsClicks` is `true`, the click mechanism is turned on. Your subclass can do whatever is necessary. Often a tool shows its floating palettes when switched in.

This method is automatically called on all tools (but not components) at startup. This method is also called when the user chooses the tool from the **Tool** menu.

switchOut

`switchOut` *self* ⇨ ??

Performs necessary actions to remove focus from this `ToolContainer`. The `theToolOrganizer` will remove this `ToolContainer`'s menus and remove the click handling. Your subclass can do whatever is necessary. Often a `ToolContainer` hides its floating palettes when switched out.

toolAbout

`toolAbout` *self* ⇨ ??

The user has requested information about the current tool by should the “About Tool” menu item in the **Tools** menu. The default `ToolContainer` method does nothing. Subclasses of `ToolContainer` should implement this method to display a dialog box containing credits, copyright information, and any other relevant information about the tool.

toolPrefs

`toolPrefs` *self* ⇨ ??

The user wants to change global settings for the current tool. The `theToolOrganizer` calls this method in response to the “Tool Preferences” item in the Tools menu. The default method does nothing. Subclasses of `ToolContainer` might want to display a dialog with various controls to change the preferences.

ToolMenuBar

Class type:	Tool class (concrete)
Resides in:	ScriptX executable but not KMP
Inherits from:	`SystemMenuBar`, `IndirectCollection`
Component:	Tools Framework

Instance Variables

parentTool

self`.parentTool`

The `ToolContainer` instance that owns the `ToolMenuBar`.

Here is how the menu process is handled on both Macintosh and Window platforms:

1. User clicks the menu bar
2. ScriptX updates its menus
3. `ToolOrganizer` calls `update` methods for each menu item in the current tool
4. Menu appears
5. User selects an item
6. ScriptX tries to handle the menu item
7. Otherwise, `theToolOrganizer` finds the menu item object and tells it to perform its action.

ToolMenu

Class type: Tool class (concrete)
Resides in: ScriptX executable but not KMP
Inherits from: `IndirectCollection`
Component: Tool Framework

A `ToolMenu` object is an array of menu items.

Instance Variables

name

self`.name` (read-write) ⇨ `String`

The `String` that is displayed in the menu bar. This variable is set as a result of the `name`: key argument to the init method. However, the value can be changed at any time. Call the `menuChanged` method in `ToolMenuBar` to update the menu bar after changing the name.

enableHandler

self`.enableHandler`

Currently unused.

Instance Methods

init

`init` *self* `name:`*string* ⇨ `ToolMenu`

self	`ToolMenu` object
`name`	`String` object

Creates a new instance of `ToolMenu`, with the given name. A tool must append this item to its `ToolMenuBar` for the menu to be installed.

ToolMenuItem

Class type: Tool class (concrete)
Resides in: ScriptX executable but not KMP
Inherits from: `RootObject`
Component: Tool Framework

A `ToolMenuItem` represents the basic unit in the menu bar. Any item that the user can actually select is an instance of a `ToolMenuItem`. Each item contains variables and methods that actually perform the display, updating, and execution of an item.

Instance Variables

name

self.`name` (read-write) `String`

A `String` object containing the text that appears in this menu item. This is first set by the `name` key argument, but can be changed at any point. Call `menuChanged` to force the menu bar to refresh.

If an item has a name of "-," then the item is drawn in the menu bar as a grayed separator line. A separator item doesn't need a `menuHandler` – it will always be dimmed and the user can never select it.

shortCut

self.`shortCut` (read-write) `String`

A one-character `String` which is used as the Command-key or Accelerator-key equivalent to this menu item. If left undefined, then the item has no key equivalent.

menuHandler

self.`menuHandler` (read-write) *(function)*

The method that gets called by the `ToolOrganizer` to update or execute the menu item. This method is called on two occasions: when the menu bar is being updated and when the user has selected this item. A `menuHandler` should have the following definition:

`menuFunc` *self theMenuItem theOptions* `-> returns object`

self can be either the tool that owns the menu item, or any arbitrary object contained in the owner instance variable.

theMenuItem is this object.

theOptions is either going to be `@update` or `@perform`.

If this handler is called with `theOptions` equal to `@update`, the `gToolOrganizer` is in the process of dimming or checking every item in the menu bar. This happens each time the user clicks on the menu bar, but before the menu is displayed. This gives the tool the opportunity to dynamically gray or check items. A `menuHandler` should return one of the following values (default is `@dimmed`):

```
@enabled
@enabledChecked
@dimmed
@dimmedChecked
@update Return Values
```

If theOptions parameter is set to @perform, then this owner object should go ahead and execute the action associated with this menu item. The return value is ignored.

If a ToolMenuItem's menuHandler is undefined, then the menu item is @dimmed.

itemNum

self.itemNum	(read-write)	Integer

Index of this item in the ToolMenu list. This is a one-based count that can be used to distinguish menu items that share a menuHandler.

itsOwner

self.itsOwner	(read-write)	Tool

Object that owns this menu item. If a Tool leaves this value undefined, the tool itself will always be the first parameter to menuHandler. Otherwise, the value of this variable is passed as the first argument to the menuHandler.

Instance Methods

init

init *self* name: *string* [shortCut: *string*] [menuFunc: *object*] [owner: *object*]

self	ToolContainer instance
name:	NameClass instance
shortCut:	Collection or String representing the path
menuFunc:	String representing the file name
owner:	LibraryContainer object

Sets up the ToolMenuItem object based on the supplied keyword arguments. They in fact correspond to the above instance variables. Only name is required.

DragRegion

Class Type:	Tool class (concrete)
Resides in:	ScriptX executable but not KMP
Inherits From:	`RootObject`
Component:	Tool Framework

The `DragRegion` class provides an interface for tools to drag and drop objects into title or tools windows. For the user, the `DragRegion` appears as a dotted gray rectangle. As the user drags the mouse across the screen, this rectangle follows. After the mouse button is released, the `DragRegion` returns the window underneath the rectangle.

Creating and Initializing a New Instance

To create an instance of `DragRegion`, call `new` on it, and supply an optional rectangle to `boundingBox`:

```
myDragRgn := new DragRegion boundingBox:(new Rect x2:100 y2:50)
```

The variable `myDragRgn` contains the initialized drag region, which has the dimensions of 100 pixels wide by 50 pixels high. The `new` method uses the keywords defined in `init`.

init

`init` *self* [`boundingBox:`*rect*] ⇨ *self*

self	`DragRegion` object
`boundingBox:`	`Rect` object representing the boundary of the region

Initializes this instance of `DragRegion`. The value supplied with the `boundingBox` keyword is saved to mthe instance variable `rBounds`. If `boundingBox` is not specified, it will not throw an exception, but the `rBounds` instance variable should be set before calling `dragMe`.

Instance Variables

dropSurf

self`.dropSurf` (read-write) `DisplaySurface`

The `dragMe` method sets this instance variable to the `DisplaySurface` object onto which the shape was dropped. If the user dropped the shape into the desktop, then this value is set to undefined.

dropX

self`.dropX` (read-write) `Number`

The x offset of the dropped shape, as set by the `dragMe` method. This value is provided in local coordinates of the `dropSurf` value. If the shape was not dropped in a `DisplaySurface`, then this value of this field should be ignored.

dropY

self`.dropY` (read-write) `Number`

The y offset of the dropped shape, as set by the `dragMe` method. This value is provided in local coordinates of the `dropSurf` value. If the shape was not dropped in a `DisplaySurface`, then this value of this field should be ignored.

rBounds

self.rBounds

Contains the Rect object that forms the outline. The x1 and y1 fields of the rBounds object are significant – they provide the initial position of the drag rectangle.

xGrid

self.xGrid

The number of pixels to which to grid the horizontal dragging. Uses the xOrigin as the origin of the grid. *Not implemented in current release.*

xOrigin

self.xOrigin

Specifies the number of pixels in screen space to offset the grid in the horizontal direction. *Not implemented in current release.*

yGrid

self.yGrid

The number of pixels to which to grid the vertical dragging. Uses the yOrigin as the origin of the grid. *Not implemented in current release.*

yOrigin

self.yOrigin

Specifies the number of pixels in screen space to offset the grid in the vertical direction. *Not implemented in current release.*

Instance Methods

dragMe

dragMe *self* ⇨ Boolean

Call this method in response to a mouse click in the drag region's bounding box. This creates a gray rectangle underneath the mouse, which the user drags around until he or she releases the mouse button. If the mouse was released on top of a valid window, then this method returns true and sets the dropSurf, dropX, and dropY instance variables. Otherwise, it returns false.

CHAPTER

14

Photoshop Plug-ins for KIC Compression

Kaleida Labs has developed a new algorithm for compressing images that meets the needs of multimedia title developers. Users can save images using this new compression format by using a set of plug-ins to Adobe Photoshop™.

These plug-ins enable Adobe Photoshop to save or export images in Kaleida Labs' new compression format, KIC (Kaleida Image Format). Compressed image files generated by these plug-ins can then be imported into ScriptX using the new KIC importer. The resulting compressed ScriptX bitmaps can be saved out to the object store and manipulated in the same manner as any other ScriptX `Bitmap` object.

What is KIC?

The KIC compression algorithm has the following features:

- Maintains high quality for both synthetic and natural scene images.
- Achieves good (near lossless for synthetic images) quality compression at ~2 bits per pixel.
- Is quick to decompress, since it decompresses pixels and draws them directly to their target location, rather than first drawing them to an offscreen buffer.
- Can do partial decompression, which is useful for animation purposes where only a portion of an image may need to be redrawn. This reduces RAM requirements for bitmaps, since they can be stored compressed in memory and decompressed partially or completely directly to the screen. Partial decompression is discussed in more detail later.
- Handles invisible color during decompression, which can speed up sprite animation that makes extensive use of invisible colors. The handling of invisible colors is discussed in more detail later.

The KIC compression algorithm works with RGB color only, so if you want to save a gray scale picture to KIC, convert it to RGB color first.

Partial Decompression

During the compositor update cycle in ScriptX, each presenter redraws the parts of the thing it is presenting that have changed. For example, if a window that is partially hidden by another window is brought to the front, only the parts of the window that were previously obscured need to be redrawn. For most decompression algorithms, if a bitmap needs to be redrawn, the entire bitmap is decompressed and drawn to an offscreen buffer, then the parts of the bitmap that have changed are drawn to the onscreen target. However, the KIC decompression algorithm decompresses only the parts of the bitmap that have changed, and draws them directly to the target, skipping the step of drawing to the offscreen buffer.

Invisible Color

The invisible color of a bitmap is a color that does not get drawn, giving the effect that the parts of the bitmap that use that color are transparent. For most decompression algorithms, when the bitmap is decompressed all pixels are drawn to an offscreen buffer, including those pixels that use the invisible color. When the compositor transfers the image from the offscreen buffer to the target on screen, the pixels that use the invisible color are not drawn. However, the KIC decompression algorithm detects pixels that use the invisible color during decompression, and does not draw them at all.

Where Can the Plug-ins Be Used?

The KIC compression algorithm is available through plug-ins for Adobe Photoshop, and other applications that can work with Photoshop plug-ins. These plug-ins allow you to save images in KIC format, either by saving or exporting images.

For the Macintosh, file-format and export plug-ins for KIC are available. The file-format plug-ins, which allow you to save images as KIC, work in Photoshop 3.0. The export plug-ins, which allow you to export images as KIC, work in Photoshop 2.0 and later. There are two versions of each plug-in file, one for Macintoshes that have a 68881 math co-processor and one for all other platforms on which ScriptX runs, including Macintosh computers that do not have a 68881 math co-processor. (Exporting and saving images as KIC is much faster on Macintoshes that have a 68881 co-processor if you use the corresponding plug-in.)

For Windows systems, a file-format plug-in is available which works in Photoshop 3.0.

Installing the Plug-ins

To install the KIC plug-ins on a Macintosh, put the files in the Plug-Ins folder for your Adobe Photoshop application.

If your Macintosh uses a 68881, put the `KICFormat` and `KICExport` files in the Plug-Ins folder for Adobe Photoshop. (The version that requires the 68881 will put up a dialog box if it detects that there isn't a 68881 installed.)

If your Macintosh does not use a 68881 math co-processork, put the files `KICFormat no 68881` and `KICExport no 68881` in the plug-ins folder.

(All Macintosh Quadras and AVs have math co-processors. Many of the lower-end Macs (LCs and Centris's) have them as an optional add-on.)

On a Windows machine, put the file `KICFrmt.8bi` in the Plug-Ins folder for Adobe Photoshop.

Using the Plug-ins

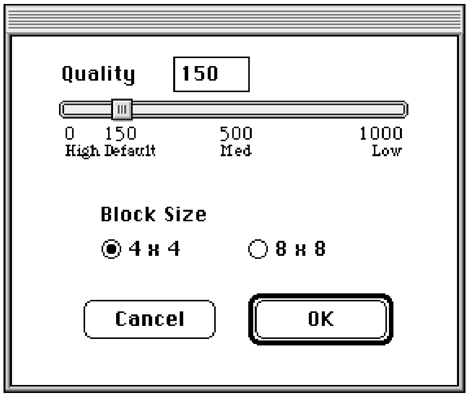
If the appropriate KIC plug-in file has been put in the Plug-Ins folder for Adobe Photoshop 3.0, then when you choose the **Save As** menu command in PhotoShop, one of the available file types will be **KIC Compression**. If you choose this option, a dialog box appears as shown in Figure 14-1.

KIC will also be one of the file types offered by the **Open** menu command.

If the the appropriate KIC plug-in file has been put in the Plug-Ins folder for Adobe Photoshop 2.5 or later, then when you choose the **Export** menu command in Photoshop, one of the available file types will be **KIC Compression**. (Macintosh only.) If you choose this option, a dialog box appears as shown in Figure 14-1.

The **Qualitity** slider (on some platforms this may appear as a scrolbar) let's you determine the quality of the compressed image as determined. 0 indicates highest quality, and 1000 indicates lowest quality. The higher the quality, the less the compression, and hence the larger the resulting file.

The **Block Size** radio buttons let you determine whether the compression algorithm uses 4x4 or 8x8 blocks. If 4x4 block size is used, the compression will be better and the resulting file size will be larger than if 8x8 is used. For images that don't use many colors, however, 8x8 often provides almost as good quality as 4x4, which significant decrease in file size.



Quality 150

Block Size

◉ 4 x 4 ○ 8 x 8

Figure 14-1: The Dialog Box for Saving As KIC compression in Adobe Photoshop

The Playback Side

To use images compressed in the KIC format in ScriptX, import them using the KIC importer. The decompression side of the KIC algorithm is available in ScriptX as a ScriptX codec (similar to the Cinepak codec). The ScriptX graphics system has already been modified to take advantage of the features of individual codecs. Hence an author can easily to take advantage of the partial decompression and invisible color facilities of the decompressor. The hope is that authors can use the codec for both background images and animated sprites. The result will be reduced disk-based and in-memory footprints without a performance loss.

CHAPTER

Importing Media

15

This chapter describes the Import/Export Tool in ScriptX. This tool allows you to import files created outside of ScriptX into ScriptX. The Import/Export Engine translates data in non-ScriptX files into ScriptX objects during importing and conversely translates ScriptX objects to other data formats when exporting.

For example, when you import an AIFF sound file into ScriptX, it becomes a `DigitalAudioPlayer` object with an associated `AIFFStream`. When you import a PICT file it becomes a `Bitmap` object, and so on.

The current release of ScriptX contains importers for text, image, sound, and movie data. Additional importers may be added for later releases.

File Organization for Importers

All importer and exporter modules (generally implemented as ScriptX extension classes), must be placed in the `importrs` directory (or folder). The `importrs` directory must be located in the same directory as the ScriptX executable. It contains a subdirectory named for the platform you are using (for example, `mac` , `win`, or `os2`). Each platform subdirectory contains the loadable extensions that implement the import/export capabilities.

When the ScriptX development environment starts up, it searches the `importrs` directory for extensions, and loads the ones that are appropriate to the platform and current software version. The code for the importer and exporter routines is not loaded; instead, the kinds of conversions supported by the importer and exporter modules are registered with the ScriptX system. The import/export engine loads the code for a specific conversion only when it is needed, not before.

Importing

To import a non-ScriptX file into Scriptx, call the `importMedia` method on `theImportExportEngine` global object.This object is an instance of the class `ImportExportEngine.`

importMedia

`importMedia theImportExportEngine` *source dataCategory* \
dataFormat outputType ⇨ (*object*)

source	The source of the data to be imported. This is usually expressed as a `Stream` object, although in some cases it can be a `Collection` of strings that describe the pathname to the file.
dataCategory	A name token indicating the category of data to be imported.
dataFormat	A name token indicating the format of the data in the input stream.
outputType	A name token indicating the type of object created in ScriptX to hold the imported data.

Additional parameters required by a specific importer can be passed using keyword arguments.

The options supported by the different importers are listed in the section for each importer later on in this chapter.

The *source* is usually a `Stream` object that points to a file containing the data to be imported. In the case of importing flattened, QuickTime, Cinepak-compressed movies, *source* must be a `Collection` of strings that make up the path name of the file.

To create a stream that points to a file, use the `getStream` function, which takes three arguments: the directory containing the file, the name of the file, and a token that indicates the properties of the stream. For streams for imported files, the property is always `@readable`. You can use the global instance `theScriptDir` to indicate the directory where the running script resides.

For example, to create a stream that points to the file `lion.aif` in the directory containing this script:

```
lionstream:= getStream theScriptDir "lion.aif" @readable
```

If the importer is successful in converting the data, it returns a ScriptX object of the specified output type. The example below converts a QuickTime file named `hummbird` to a ScriptX `MoviePlayer` object.

```
mp := importMedia theImportExportEngine \
   (getStream theScriptDir "hummbird" @readable) \
   @movie @quicktime @Player
```

Available Output Types

A single importer can often produce different types of objects as output. You may choose the type of output you want depending on your needs. Often, there are two types of objects that the importer can produce, the raw media object, and a media object attached to a ScriptX player.

For example, if you are importing an audio file and you want to work with it immediately after importing, you would request that the importer generate a digital audio player. On the other hand, if you were doing a batch import of a number of audio files and were transferring them to ScriptX containers, there would be no need to generate players, you would simply request the translated media as Stream objects.

In the following examples, the importMedia method is used to convert an AIFF file into a ScriptX player and also into an audio stream. The output type is the fifth parameter to the importMedia method.

```
source := getStream theScriptDir "Lumberjack Song" @readable

-- import the sound into an audio stream
ljPlayer := importMedia theImportExportEngine source \
   @sound @aiff @digitalaudioplayer

import the sound into an audio player
ljData := importMedia theImportExportEngine source \
   @sound @aiff @audiostream
```

Specifying A Container for the Imported Data

Some of the importers accept optional keyword arguments that control different aspects of behavior of the importer. The options supported by each import module are listed in the section for each importer later on in this chapter. The most common optional keyword is the container keyword, which allows you to specify a container into which the raw data for the media object will be stored.

When an importer accepts a container keyword argument, the container is used to store the raw data for the imported media. After the importing process has finished, you still need to add an object that uses the media (or references an object that uses the media) to the title container.

For example, in the following code sample, the example from the previous section has been modified to import an AIFF stream and store it into a ScriptX container as an audio player.

```
tc := new titleContainer path:"mytitle.sxt"
ljSource := getStream theScriptDir "LumbSong.aif" @readable
ljPlayer := importMedia theImportExportEngine ljSource \
   @sound @aiff @audioplayer container:tc

-- save the audioplayer to the title container
append tc ljPlayer

-- You also need to write a startup function
-- and close the title container
```

Exporting

There is one exporter module shipping with the current release, which is a text exporter that saves text as “ScriptX Ascii” text.

The exportMedia method has five positional parameters: a destination to which the converted data is to be written, a name token describing the data’s category, the input form, the output form, and the object to be exported. Additional parameters required by a specific exporter can be passed using keyword arguments.

Text Exporter

ScriptX 1.5 has a text exporter, that enables you to export text to a file. The text will be formatted in “ScriptX Ascii.” The first line of the file will always be “ScriptX ascii\n.”

To export text, call the `exportMedia` method on `theImportExportEngine`. The arguments are the same as for `importMedia`. The stream argument specifies the file to write to.

```
exportMedia theImportExportEngine stream \
   @text @text @asciitext userData:
```

The `userData` argument is an array containing objects that are used by the text.

A “ScriptX Ascii” file contains ascii text that is marked with tags to denote ScriptX text attributes (in much the same way that rtf uses tags to denote text characteristics.)

The following few lines are extracted from the beginning of a ScriptX ascii text, to give an idea of what such a file might look like. In this case, the `&1` for the font indicates a font object, which will be referenced through the `userData` array when the text is imported or exported.

```
ScriptX ascii
\Text
\{
leading:20 alignment:center font:&1 size:18
}THE TIME MACHINE\n
```

Importers Supplied by Kaleida Labs

The following table shows the importers supplied by Kaleida Labs with the current release of ScriptX. It also identifies the media formats and the platforms on which they are supported. Currently, ScriptX runs on Macintosh, Windows, and OS/2.

Table 15-1: Importers supported in the current release

Data category argument	Data format argument	Output type argument	Resulting class	Platform
`@text`	`@RTF`	`@RichText`	`Text`	All platforms
`@text`	`@Text`	`@AsciiText`	`Text`	All platforms
`@image`	`@Pict`	`@Bitmap` `@Colormap`	`Bitmap` `Colormap`	Macintosh
`@image`	`@DIB` (See note below)	`@Bitmap or` @Compressed-Bitmap `@Colormap`	`Bitmap` `colormap`	All platforms
`@image`	`@KIC`	`@Bitmap or` @Compressed-Bitmap	`Bitmap`	All platforms

Table 15-1: Importers supported in the current release

Data category argument	Data format argument	Output type argument	Resulting class	Platform
@image	@quicktime	@Bitmap or @CompressedBitmap	Bitmap	Macintosh
		@TwoDShape	TwoDShape	
		@Presenter	TwoDShape	
@sound	@AIFF	@Stream or @AudioStream	AudioStream	All platforms
		@Player or @DigitalAudioPlayer	DigitalAudioPlayer	
@sound	@SND	@Stream or @AudioStream	AudioStream	Macintosh
		@Player or @DigitalAudioPlayer	DigitalAudioPlayer	
@sound	@WAVE (See note below)	@Stream or @AudioStream	AudioStream	All platforms
		@Player or @DigitalAudioPlayer	DigitalAudioPlayer	
@MIDI	@standard	@Stream or @MIDIstream	MIDIStream	All platforms
		@Player or @MIDIPlayer	MIDIPlayer	
@movie	@quicktime	@InterleavedMoviePlayer	InterleavedMoviePlayer	All platforms
@movie	@AVI	@InterleavedMoviePlayer	InterleavedMoviePlayer	All platforms

Note – The OS/2 DIB importer accepts any format that is supported by OS/2 via the MMPM/2 IOprocs. Likewise, the OS/2 WAVE importer accepts any format that is supported by OS/2 MMPM/2 IOprocs. See the OS/2 multimedia documentation for details.

Text Importer

ScriptX provides a text importer that can import richt text format (rtf) text, ascii text, or “ScriptX Ascii” text (which is text that has been previously exported from ScriptX.)

The line ending conventions for unix, mac, and dos are all handled correctly. The text importer translates multiple consecutive blank lines as a paragraph boundary. Blank here means any line containing just whitespace. Any 8 bit characters are translated to '?' by the plain text importer.

The 1.5 importer treats consecutive blank lines as a paragraph boundary. Put another way, the 1.1 importer fills paragraphs. This importer is 30X faster than the 1.0 importer.

ASCII, ScriptX ASCII or RTF Text to Text

importMedia

`importMedia theImportExportEngine` *stream* `@Text\`
inputType *inputType* `[userData:]` ⇨ `Text`

source — A `Stream` object specifying the file containing the data to be imported.

`@Text` — Denotes that the imported data is text.

inputType — Either `@ASCIItext` or `@rtf`. Denotes whether rtf or ASCII text is to be imported.

inputType — Denotes that a `Text` object will be created and returned by the importing process. If the value is @Ascii, then the `Text` object will contain plain, unformatted text. If the value is `@richText`, the `Text` object will contain formatted text (this option is only available for importing rtf files, not Ascii files.)

`userData:` — Is only needed for importing ScriptX Ascii text. It is an array containing objects that are used by the text.

This importer creates a `Text` object that contains the imported text.

For example, the following code shows how to import text stored in the ASCII file “`buttrfly.doc`” and display it in a `TextPresenter`: (assuming the ASCII file is in the same directory as the script file):

```
global butterflyStream := getstream theScriptDir "buttrfly.doc" \
   @readable
global butterflyText := importMedia theImportExportEngine \
   butterflyStream @Text @ASCIITExt @Text
global butterflyTextPres := new TextPresenter \
   target:butterflyText \
   boundary:(new rect x2:400 y2: 400) \
   brush:(new brush color:blueColor)
global w:= new window boundary:(new rect x2:500 y2: 500)
show w
append w butterflyTextPres
-- Center the text presenter in the window
butterflytext.x := 50
butterflytext.y := 50
```

When the text importer imports rtf text, it creates and returns a `Text` object that contains the imported text, retaining most of the the font and paragraph characteristics of the text in the original file.

The text and paragraph characteristics that the importer handles properly include:

- flush left, flush right, fill, and center alignments
- paragraph indents, left and right indents
- leading
- space between and space before
- font
- font size
- expand-condensed, expand-normal, expand-expanded
- no underline and single underline
- bold and italic typestyles
- foreground color font

The text and paragraph characterists that the importer does not handle properly include:

- background color
- superscript and subscript type styles
- double underline
- strike through, outline, shadow,
- small capitals, all capitals,
- RTF invisible, gray and deleted character styles.

Table 15-2, "RTF conversion" shows how certain characters in RTF are translated by the RTF to ScriptX importer.

Table 15-2: RTF conversion

RTF	RTF Importer in ScriptX
Page, sect, row, line,and paragraph terminators	newline '\n'.
Tab	'\t'
NoBrkHyphen	'-'
Bullet	0x2022 UNICODE for bullet
EmDash	as 0x2014 UNICODE for long dash
EnDash	0x2013 UNICODE for short dash
LQuote	0x2018 UNICODE for left curly single quote

Table 15-2: RTF conversion

RQuote	0x2019 UNICODE for right curly single quote
LDblQuote	0x201c UNICODE for left curly double quote
RDblQuote	0x201d UNICODE for right curly double quote

The following example shows how to import and display the text stored in the RTF file "snail.doc"if it was stored in the same folder as the script:

```
global snailStream := getstream theScriptDir "snail.doc" @readable
global snailText := importMedia theImportExportEngine snailstream \
   @Text @rtf @RichText
global snailStory := new textPresenter target:snailText \
   boundary:(new rect x2:400 y2: 400)
global w := new window
w.width := 500
w.height := 500
show w
append w snailstory
-- Center the text presenter in the window
snailStory.x := 50
snailStory.y := 50
```

Image importers

All image importers in the current release of ScriptX import external image data into a ScriptX `Bitmap` object.

Pict to Bitmap

importMedia

`importMedia theImportExportEngine` *source* `@image @pict` *outputType*
`[colormap:` *colormap*`]` ⇨`Bitmap` or `Colormap`

source — A `Stream` object specifying the file containing the data to be imported.

`@image` — Denotes that the imported data is an image.

`@pict` — Denotes that the format of the imported data is pict.

outputType — Denotes the class of the object created to hold the imported data. The choices are:

- `@bitmap` — The object created to hold the imported data is a `Bitmap`.
- `@colormap` — The object created to hold the imported data is a `ColorMap`. When the output is specified as `@colormap`, only the palette information is read from the PICT.

`colormap:` *colormap* — If this optional keyword argument is used, *colormap* must be a ScriptX `ColorMap` object. The imported bitmap will be constructed using the supplied `ColorMap` object.
You probably don't want to use the `colormap` keyword if the *outputType* is `@colormap`.

This importer returns a ScriptX `Bitmap` object upon successfully converting a Macintosh PICT or PICT2 image. The source file to be imported may be a data file in the PICT file format, or a PICT resource in any Macintosh file.

The input picture may be any resolution and any color level. The resulting bitmap will be of the same bit-depth as the input picture if no `colormap` keyword is specified, otherwise it will be at the bit-depth of the given colormap.

For example, to import a pict file called "lobster pict" and to display the lobster in ScriptX (assuming the pict file is in the same folder as the script file):

```
global lobfile := getStream theScriptDir "lobster pict" @readable
global lobbie := importMedia theImportExportEngine lobfile \
   @image @pict @bitmap

-- Create and show a window
global w:= new window
show w

-- Make a twoDshape for the lobster
-- Show the lobster in the window
global lobPresenter := new TwoDShape boundary: lobbie
append w lobPresenter
```

The following code shows an example of importing a pict from a Macintosh resource file:

```
global crawBundle := new resBundle dir:theScriptDir path:"crawdad.res"
global crawStream := getOneStream thebundle type:"PICT" \
   name:"crawdad.pict"
global crawdaddie := importMedia theImportExportEngine crawStream \
   @image @pict @bitmap

-- Create and show a window
global w:= new window
show w

-- Make a twoDshape for the crawdad
-- Show the crawdad in the window
global crawPresenter := new TwoDShape boundary: crawdaddie
append w crawPresenter
```

DIB to Bitmap

importMedia

`importMedia theImportExportEngine` *source* `@image @dib` *outputType*
   [`colormap`: *colormap*] [`container`: *container*]
   [`pagingMethod:`*decompMethod*] ⇨ `Bitmap` or `Colormap`

source — A `Stream` object specifying the file containing the data to be imported.

`@image` — Denotes that the imported data is an image.

`@dib` — Denotes that the format of the imported data is DIB.

outputType — Denotes the class of the object created to hold the imported data. The choices are:

- `@bitmap` — The object created to hold the imported data is a `Bitmap`. If the bitmap to be imported is compressed, it is decompressed during importing.
- `@colormap` — The object created to hold the imported data is a `ColorMap`. When the output is specified as `@colormap`, only the palette information is read from the file.
- `@compressedBitmap` — The object created to hold the imported data is a `Bitmap`. If the bitmap to be imported is compressed, the existing compression is maintained during importing.

`colormap:` *colormap*

If this optional keyword argument is used, *colormap* must be a ScriptX `ColorMap` object. The imported bitmap will be constructed using the supplied `ColorMap` object.

`container`: *container*

If this optional keyword argument is used, its value must be a`LibraryContainer, TitleContainer` or `AccessorContainer` object that is used to save imported, compressed bitmaps. This keyword argument is only needed when the output type is specified as `@compressedBitmap`. If you plan to save a compressed bitmap to a title container, you must supply the container keyword argument when you do the importing.

pagingMethod: *nameClass*

The value of this keyword argument determines how and when a compressed bitmap is decompressed. This argument is only needed if the output type is specified as @compressedBitmap. The choices are:

@onload The bitmap is decompressed when it is loaded from the object store and remains decompressed.

@firstUse The bitmap is decompressed the first time it is used, and remains decompressed.

@eachUseFromStorage

The bitmap is loaded from the object store and decompressed each time it is used. The data is cleared from memory after each use. This is the default value.

@eachUseFromMemory

The bitmap is loaded and decompressed the first time it is used. The compressed data stays in memory. The bitmap is decompressed each time it is used again.

This importer returns a ScriptX Bitmap object if it succeeds in converting an image in the Windows DIB (Device-Independent Bitmap) format to a ScriptX Bitmap object.

The input picture can be 1-bit, 4-bit, 8-bit or 24-bit, and may be uncompressed or compressed in RLE4 or RLE8 format.

On Windows, the resulting bitmap has the same bit depth as the computer that you are using. On the Macintosh, the resulting bitmap has the same bit depth as the input picture if no color map is specified, otherwise it has the same bit depth as the color map. OS/2.

If the DIB file containing the input picture is uncompressed, then it is imported uncompressed. If it is compressed, you can choose whether to import it uncompressed or compressed. To decompress it during importing, specify the output type as @bitmap. To maintain the compression during importing, specify the output type as @compressedBitmap, and supply a storage container for the container keyword argument.

If you want to save an uncompressed bitmap to a title container, you don't need to specify a container when importing it. Simply append the Bitmap object (or any object that refers to it) returned by the importing process to a title container.

However, if you want to save a compressed bitmap to a title container, you must provide a library, title, or accessory container as the container keyword argument at import time, and also append the Bitmap object returned by the importing process (or any object that refers to it) to the title container.

The value of the pagingMethod keyword argument determines when the compressed bitmap data is decompressed. This value is stored in the Bitmap object's pagingMethod instance variable.

When the bitmap data is imported, the raw, compressed data is saved to the title container provided for the `container` keyword argument. The decompression method determines when the bitmap is decompressed when it is loaded back from the object store. If you use the bitmap before saving it to the object store, the compressed data is still saved when and if you eventually save it. .

If the `pagingMethod` keyword argument is `@onload`, the bitmap is decompressed when it is loaded from the object store, and the decompressed data is held in memory.

If the `pagingMethod` keyword argument is `@firstUse`, the bitmap is decompressed when it is first used, and the decompressed data is held in memory. .

If the `pagingMethod` keyword argument is `@eachUseFromStorage`, (which is the default) the bitmap is fetched from the title container and decompressed each time it is used. Neither the decompressed or compressed data is kept in memory.

If the `pagingMethod` keyword argument is `@eachUseFromMemory`, the bitmap is decompressed each time it is used. The compressed data is held in memory.

The following code shows how to import a dib file called "jelly.dib" and to display the jellyfish in ScriptX: (assuming the dib file is in the same directory as the script file)

```
global jellyfishfile := getStream theScriptDir "jelly.dib" @readable
global jellie := importMedia theImportExportEngine jellyfishfile \
    @image @dib @bitmap \

-- Create and show a window
global w:= new window
show w

-- Make a twoDshape for the jellyfish
-- Show the jellyfish in the window
global jellyfishPresenter := new TwoDShape boundary: jellie
append w jellyfishPresenter
```

The following example shows the code modifed to import a compressed bitmap and save it to a title container.

```
global oceantc := new titlecontainer path:"ocean.sxt"
global jellyfishfile := getStream theScriptDir "jelly.dib" @readable
global jellie := importMedia theImportExportEngine jellyfishfile \
    @image @dib @bitmap \
    container:oceantc \
    pagingMethod:@eachUseFromMemory

-- Create and show a window
global w := new window
show w

-- Make a twoDshape for the jellyfish
-- Show the jellyfish in the window
global jellyfishPresenter := new TwoDShape boundary: jellie
append w jellyfishPresenter

-- Save the window to the title container
append oceantc jellyfishPresenter

-- You’d also need to write a startup function and
-- close the title container
```

KIC to Bitmap

importMedia

`importMedia theImportExportEngine` *source* `@image @KIC` *outputType*
[`container`: *container*] [`pagingMethod:`*decompMethod*] ⇨ `Bitmap`

source
A `Stream` object specifying the file containing the data to be imported.

`@image`
Denotes that the imported data is an image.

`@KIC`
Denotes that the format of the imported data is KIC

outputType
Denotes the class of the object created to hold the imported data. The choices are:

- `@bitmap` The object created to hold the imported data is a `Bitmap` that holds compressed data.
- `@CompressedBitmap` Same as `@bitmap`.

`container`: *container*
If this optional keyword argument is used, its value must be a`LibraryContainer, TitleContainer` or `AccessorContainer` object that is used to save imported, compressed bitmaps. If you plan to save a compressed bitmap to a title container, you must supply the `container` keyword argument when you do the importing.

`pagingMethod`: *nameClass*
The value of this keyword argument determines how and when a compressed bitmap is decompressed. The choices are:

- `@onload` The bitmap is decompressed when it is loaded from the object store and remains decompressed.
- `@firstUse` The bitmap is decompressed the first time it is used, and remains decompressed.
- `@eachUseFromStorage` The bitmap is loaded from the object store and decompressed each time it is used. The data is cleared from memory after each use. This is the default value.
- `@eachUseFromMemory` The bitmap is loaded and decompressed the first time it is used. The compressed data stays in memory. The bitmap is decompressed each time it is used again.

This importer imports images saved in the KIC (Kaleida Image Compression) format. If the importing is successful, it returns a `Bitmap` object containing compressed data.

If you want to save the imported, compressed bitmap to a title container, you must provide a library, title, or accessory container as the `container` keyword argument at import time, and also append the `Bitmap` object returned by the importing process (or any object that refers to it) to the title container.

The value of the pagingMethod keyword argument determines when the compressed bitmap data is decompressed. This value is stored in the Bitmap object's pagingMethod instance variable.

When the bitmap data is imported, the raw, compressed data is saved to the title container provided for the container keyword argument. The decompression method determines when the bitmap is decompressed when it is loaded back from the object store. If you use the bitmap before saving it to the object store, the compressed data is still saved when and if you eventually save it. .

If the pagingMethod keyword argument is @onload, the bitmap is decompressed when it is loaded from the object store, and the decompressed data is held in memory.

If the pagingMethod keyword argument is @firstUse, the bitmap is decompressed when it is first used, and the decompressed data is held in memory.

If the pagingMethod keyword argument is @eachUseFromStorage, (which is the default) the bitmap is fetched from the title container and decompressed each time it is used. Neither the decompressed or compressed data is kept in memory.

If the pagingMethod keyword argument is @eachUseFromMemory, the bitmap is decompressed each time it is used. The compressed data is held in memory.

The following code imports an KIC file called "door.KIC," displays the door in ScriptX:, and then saves it to a title container. (assuming the door file is in the same directory as the script file).

```
global doortc := new titlecontainer path:"door.sxt"
global doorfile := getStream theScriptDir "door.KIC" @readable
global door := importMedia theImportExportEngine doorfile \
    @image @KIC @compressedbitmap container:doortc \
    pagingMethod:@eachUseFromMemory

-- Create and show a window
global w := new window
show w

-- Make a twoDshape for the door
-- Show the door in the window
global doorPresenter := new TwoDShape boundary: door
append w doorPresenter

-- Save the window to the title container
append doortc doorPresenter

-- You'd also need to write a startup function and
-- close the title container
```

QuickTime to Bitmap

importMedia

`importMedia theImportExportEngine` *source* `@image @quicktime`
outputType [`frame`: *frameNumber*] `container`: *container*
[`pagingMethod:`*decompMethod*] ⇨ `Bitmap` or `TwoDShape`

source — A `Stream` object specifying the file containing the data to be imported.

`@image` — Denotes that the imported data is an image.

`@quicktime` — Denotes that the format of the imported data is QuickTime.

outputType — Denotes the class of the object created to hold the imported data. The choices are:

- `@bitmap` — The object created to hold the imported data is a `Bitmap` that holds compressed data.
- `@CompressedBitmap` — Same as `@bitmap`.
- `@TwoDShape` — The object created to hold the imported data is a `TwoDShape`.
- `@Presenter` — Same as `@TwoDShape`.

`frame:` *frameNumber*

Denotes the frame to import. This frame will be converted to a `Bitmap` object. This frame must be a key frame of the movie. (See the discussion below for more details on key frames.)

`container`: *container*

If this optional keyword argument is used, its value must be a`LibraryContainer,` `TitleContainer` or `AccessorContainer` object that is used to save imported, compressed bitmaps. If you plan to save a compressed bitmap to a title container, you must supply the `container` keyword argument when you do the importing.

`pagingMethod`: *nameClass*

The value of this keyword argument determines how and when a compressed bitmap is decompressed. The choices are:

- `@onload` — The bitmap is decompressed when it is loaded from the object store and remains decompressed.
- `@firstUse` — The bitmap is decompressed the first time it is used, and remains decompressed.
- `@eachUseFromStorage` — The bitmap is loaded from the object store and decompressed each time it is used. The data is cleared from memory after each use. This is the default value.
- `@eachUseFromMemory`

The bitmap is loaded and decompressed the first time it is used. The compressed data stays in memory. The bitmap is decompressed each time it is used again.

This importer converts the specified key frame in the QuickTime file to a `Bitmap` object, and returns the bitmap, or a `TwoDShape` whose stencil is the bitmap, depending on the value of the *outputType* argument. The importer maintains compression of the QuickTime bitmaps.

The intent of this importer is to import a single frame from a QuickTime file and convert it to a bitmap. You can theoretically import a frame from any QuickTime file, but the importer is intended for use with Cinepak compressed movies that contain only key frames.

The imported frame must be a key frame. When QuickTime data is compressed, certain compression algorithms work by saving the data for a whole frame (a key frame), then for a certain number of following frames, saving only the incremental differences between the frames. The QuickTime to Bitmap importer can only import key frames, that is, frames for which all the data has been saved.

Use this importer with QuickTime files for which you know which frames are the key frames. The `frame` keyword argument requires the actual frame number. For example, suppose frame 1 is a key frame and frame 10 is the next key frame. To import the second key frame, specify the `frame` keyword argument as 10 (not 2).

If you want to save the imported, compressed bitmap to a title container, you must provide a library, title, or accessory container as the `container` keyword argument at import time, and also append the `Bitmap` or `TwoDShape` object returned by the importing process (or any object that refers to it) to the title container.

The value of the `pagingMethod` keyword argument determines when the compressed bitmap data is decompressed. This value is stored in the Bitmap object's `pagingMethod` instance variable.

When the bitmap data is imported, the raw, compressed data is saved to the title container provided for the `container` keyword argument. The decompression method determines when the bitmap is decompressed when it is loaded back from the object store.

If the `pagingMethod` keyword argument is `@onload`, the bitmap is decompressed when it is loaded from the object store, and the decompressed data is held in memory. .

If the `pagingMethod` keyword argument is `@firstUse`, the bitmap is decompressed when it is first used, and the decompressed data is held in memory.

If the `pagingMethod` keyword argument is `@eachUseFromStorage`, (which is the default) the bitmap is fetched from the title container and decompressed each time it is used. Neither the decompressed or compressed data is kept in memory.

If the `pagingMethod` keyword argument is `@eachUseFromMemory`, the bitmap is decompressed each time it is used. The compressed data is held in memory.

The following code illustrates how to import the fifth frame, which is a key frame, from the QuickTime Cinepak compressed file "horses" and display the resulting bitmap in ScriptX (assuming the file is in the same directory as the script file). The code also shows how to save the `TwoDshape` object displaying the bitmap to a title container.

```
global tc := new titleContainer path:"horse.sxt"

global horsesStream := getStream theScriptDir "horses" @readable
global horsePic := importMedia theImportExportEngine horseStream \
   @image @quicktime @twoDShape frame:5 \
   container:tc pagingMethod:@onload

-- Create and show a window
-- Display the horse picture in the window
global w:= new window; show w
append w horsePic

-- append the twoDshape to the title container
append tc horsePic

-- don’t forget to write a startup function and
-- close the title container
```

Sound importers

All sound importers in the current release of ScriptX import external data formats into a ScriptX `AudioStream` or a `DigitalAudioPlayer` object. These objects are based on the AIFF data format, and support stereo sound sampled at rates up to 22KHz.

AIFF to AudioStream

importMedia

`importMedia theImportExportEngine` *source* `@sound @AIFF`
outputType `[container:` *container*`]`⇨ `DigitalAudioPlayer`

source — A `Stream` object specifying the file containing the data to be imported.

`@sound` — Denotes that the imported data is a sound.

`@AIFF` — Denotes that the format of the imported data is AIFF.

outputType — Denotes the class of the object created to hold the imported data. The choices are:

- `@stream` — The object created to hold the imported data is an `AudioStream`.
- `@audioStream` — Same as `@stream`.
- `@player` — In this case, an `AudioStream` object is created to hold the imported sound, and a `DigitalAudioPlayer` is created with the `AudioStream` as its media stream. The `DigitalAudioPlayer` is returned by `importMedia`.
- `@digitalAudioPlayer` — Same as `@player`.

`container`: *container*

If this optional keyword argument is used, its value must be a`LibraryContainer, TitleContainer` or `AccessorContainer` object that is used to save imported audio data. If you plan to save imported audio to a title container, you must supply the `container` keyword argument when you do the importing. The raw data for the sound will be stored in this container.

This importer creates a ScriptX `AudioStream` object to hold the imported sound data. The returned object is either the `AudioStream` or a `Player` that plays the `AudioStream`. The input file must be a file in the AIFF format that is not compressed.

If you want to save the audio stream or player to a title container, you must supply a container as the `container` keyword argument. This container is used to store a stream containing only the raw data for the media. You must additionally append the `AudioStream` or `DigitalAudioPlayer` object, or any object that refers to them, to the title container when the importing process is finished.

For example, to import and play the tune stored in the file “Bell.AIF” (assuming the AIFF file is in the same directory as the script file):

```
global bellstream := getstream theScriptDir "Bell.AIF" @readable
global bellplayer := importMedia theImportExportEngine bellstream \
   @sound @AIFF @player
playprepare bellplayer 1
play bellplayer
```

In the following example, the above code has been modified to save the `DigitalAudioPlayer` to a title container.

```
global tc := new titleContainer path:"mytitle.sxt"
global bellstream := getstream theScriptDir "Bell.AIF" @readable
global bellplayer := importMedia theImportExportEngine bellstream \
   @sound @AIFF @player container:tc
playprepare bellplayer 1
play bellplayer

-- save the player to the title container
append tc bellPlayer

-- you also need to write a startup action
-- and save the title container
```

SND to AudioStream

importMedia

`importMedia theImportExportEngine` *stream* `@sound @snd` *output*
`[container:` *container*`]`⇨ `AudioStream` or `DigitalAudioPlayer`

stream — A `Stream` object specifying the file containing the data to be imported.

`@sound` — Denotes that the imported data is a sound.

`@snd` — Denotes that the format of the imported data is SND.

outputType — Denotes the class of the object created to hold the imported data. The choices are:

- `@stream` — The object created to hold the imported data is an `AudioStream`.
- `@audioStream` — Same as `@stream`.
- `@player` — In this case, an `AudioStream` object is created to hold the imported sound, and a `DigitalAudioPlayer` is created with

the AudioStream as its media stream. The DigitalAudioPlayer is returned by importMedia.

@digitalAudioPlayer

Same as @player.

container: *container*

If this optional keyword argument is used, its value must be a LibraryContainer, TitleContainer or AccessorContainer object that is used to save imported audio data. If you plan to save imported audio to a title container, you must supply the container keyword argument when you do the importing. The raw data for the sound will be stored in this container.

This importer creates a ScriptX AudioStream object to hold the imported sound data. The returned object is either the AudioStream or a Player that plays the AudioStream. The imported sound data must be in a file in the Macintosh SND format. Typically, this is an 'snd' resource stream. It must not be compressed.

If you want to save the audio stream or player to a title container, you must supply a container as the container keyword argument. This container is used to store a stream containing only the raw data for the media. You must additionally append the AudioStream or DigitalAudioPlayer object, or any object that refers to them, to the title container when the importing process is finished.

For example, to import and play the tune stored in the file "Lion.SND" (assuming the snd file is in the same directory as the script file):

```
global lionbundle := new resBundle dir:theScriptDir path:"lion.SND"
global lionstream := getOneStream lionbundle type:"snd " id:128
global lionplayer := importMedia theImportExportEngine \
    lionstream @sound @snd @player
play lionplayer
```

In the following example, the above code has been modified to save the DigitalAudioPlayer to a title container.

```
global tc := new titleContainer path:"mytitle.sxt"
global lionbundle := new resBundle dir:theScriptDir path:"lion.SND"
global lionstream := getOneStream lionbundle type:"snd " id:128
global lionplayer := importMedia theImportExportEngine \
    lionstream @sound @snd @player container:tc
play lionplayer

-- save the player to the title container
append tc lionPlayer

-- you also need to write a startup action
-- and save the title container
```

WAVE to AudioStream

importMedia

`importMedia theImportExportEngine` *stream* `@sound @wave` *output*
`[container:` *container*`]`⇨ `AudioStream` or `DigitalAudioPlayer`

source — A `Stream` object specifying the file containing the data to be imported.

`@sound` — Denotes that the imported data is a sound.

`@wave` — Denotes that the format of the imported data is WAVE.

outputType — Denotes the class of the object created to hold the imported data. The choices are:

- `@stream` — The object created to hold the imported data is an `AudioStream`.
- `@audioStream` — Same as `@stream`.
- `@player` — In this case, an `AudioStream` object is created to hold the imported sound, and a `DigitalAudioPlayer` is created with the `AudioStream` as its media stream. The `DigitalAudioPlayer` is returned by `importMedia`.
- `@digitalAudioPlayer` — Same as `@player`.

`container`: *container* — If this optional keyword argument is used, its value must be a `LibraryContainer`, `TitleContainer` or `AccessorContainer` object that is used to save imported audio data. If you plan to save imported audio to a title container, you must supply the `container` keyword argument when you do the importing. The raw data for the sound will be stored in this container.

This importer creates a ScriptX `AudioStream` object to hold the imported sound data. The returned object is either the `AudioStream` or a `Player` that plays the `AudioStream`. The file to be imported must be in the Windows WAV file format. It must not be compressed.

If you want to save the audio stream or player to a title container, you must supply a container as the `container` keyword argument. This container is used to store a stream containing only the raw data for the media. You must additionally append the `AudioStream` or `DigitalAudioPlayer` object, or any object that refers to them, to the title container when the importing process is finished.

For example, to import and play the tune stored in the file “song.wav”: (assuming the wav file is in the same directory as the script file).

```
global songstream := getstream theScriptDir "song.wav" @readable
global songplayer := importMedia theImportExportEngine \
    songstream @sound @WAVE @player
playprepare songplayer 1.0
play songplayer
```

In the following example, the above code has been modified to save the `DigitalAudioPlayer` to a title container.

```
global tc := new titleContainer path:"mytitle.sxt"
global songstream := getstream theScriptDir "song.wav" @readable
global songplayer := importMedia theImportExportEngine \
   songstream @sound @WAVE @player container:tc
playprepare songplayer 1.0
play songplayer

-- save the player to the title container
append tc songPlayer

-- you also need to write a startup action
-- and save the title container
```

MIDI Importers

The MIDI importer converts a standard MIDI file (type 0) to a ScriptX `MIDIStream` object.

Standard MIDI to Stream

importMedia

`importMedia theImportExportEngine` *stream* `@MIDI @standard` *outputType*
`[container:` *container*`]`⇨ `MIDIStream` or `MIDIPlayer`

source — A `Stream` object specifying the file containing the data to be imported.

`@MIDI` — Denotes that the data to be imported is a MIDI sound.

`@standard` — Denotes that the data to be imported is in standard MIDI format.

outputType — Denotes the class of the object created to hold the imported data. The choices are:

- `@stream` — The object created to hold the imported data is a `MIDIStream`.
- `@MIDIStream` — Same as `@stream`.
- `@player` — In this case, a `MIDIStream` object is created to hold the imported sound, and a `MIDIPlayer` is created with the `MIDIStream` as its media stream. The `MIDIPlayer` is returned by `importMedia`.
- `@MIDIPlayer` — Same as `@player`.

`container`: *container* — If this optional keyword argument is used, its value must be a `LibraryContainer`, `TitleContainer` or `AccessorContainer` object that is used to save imported MIDI data. If you plan to save imported MIDI to a title container, you must supply the `container` keyword argument when you do the importing. The raw data for the media will be stored in this container.

This importer creates a ScriptX `MIDIStream` object to hold the imported sound data. The returned object is either the `MIDIStream` or a `MIDIPlayer` that plays the `MIDIStream`.

You must have a MIDI-capable machine to import and play MIDI files.

If you want to save the midi stream or player to a title container, you must supply a container as the `container` keyword argument. This container is used to store a stream containing only the raw data for the media. You must additionally append the `MIDIStream` or `MIDIPlayer` object, or any object that refers to them, to the title container when the importing process is finished.

For example, to import and play the tune stored in the MIDI file “harp.mid”, (assuming the MIDI file is in the same directory as the script file):

```
global harpStream := getStream theScriptDir "harp.mid" @readable
global harpPlayer := importMedia theImportExportEngine harpStream \
   @MIDI @standard @MIDIPlayer
```

```
play harpPlayer
```

In the following example, the above code has been modified to save the `MIDIPlayer` to a title container.

```
global tc := new titleContainer path:"mytitle.sxt"

global harpStream := getStream theScriptDir "harp.mid" @readable
global harpPlayer := importMedia theImportExportEngine harpStream \
   @MIDI @standard @MIDIPlayer container:tc
play harpPlayer

-- save the player to the title container
append tc harpPlayer

-- you also need to write a startup action
-- and save the title container
```

Movie Importers

The Movie Importer allows you to import AVI or QuickTime movies on either a Macintosh or Windows computer. The resulting `MoviePlayer` or `InterleavedMoviePlayer` can be saved to a title container and played back on any platform that supports ScriptX.

Note – ScriptX 1.5 can import AVI and QuickTime containing any form of compressed video. ScriptX cannot import movies containing compressed audio of any sort. ScriptX supports playback of Cinepak-compressed video on all platforms. Video that was compressed with other techniques may play back on some platforms (such as the Macintosh) but there is no guarantee of portability.

QuickTime Or AVI to MoviePlayer or InterleavedMoviePlayer

importMedia

`importMedia theImportExportEngine` *source* `@movie @quicktimeOrAVI`
outputClass `[container:` *container*`] [outputClass:` *otherOutputClass*`]`
`[copydate :`*boolean*`]`

⇨ `Player`

source — A `Stream` object or a `Collection` of strings that denotes the file containing the data to be imported.

`@movie` — Denotes that the imported data is a movie.

`@quicktimeOrAVI` — Denotes that the format of the imported data. The choices are:

- `@quickTime` The file to be imported is a QuickTime file.
- `@AVI` The file to be imported is an AVI file.

outputClass — Denotes the class of the object created to hold the imported data. The choices are:

- `@Player` The object created to hold the imported data is a `MoviePlayer`.
- `@MoviePlayer` The object created to hold the imported data is a `MoviePlayer`.
- `@InterleavedMoviePlayer` The object created to hold the imported data is a `MoviePlayer`.

`container:` *container*

If this optional keyword argument is used, its value must be a `LibraryContainer`, `TitleContainer` or `AccessorContainer` object that is used to save imported movie data. If you plan to save imported movies to a title container, you must supply the `container` keyword argument when you do the importing. The raw data for the media streams containing the video and sound data will be stored in the specified container.

`copydata`: *boolean*

Use this argument if the *outputClass* is `@InterleavedMoviePlayer`.

If `copydata` is `true`, the imported media data is copied into ScriptX.This is the default. If `false`, The importer does not copy the data in, but sets up the `InterleavedMoviePlayer` so that its `interleavedStream` instance variable contains a machine specific byte stream that points to the external movie file. If `copydata` is set to false, you must take additional steps when saving the imported interleaved movie player to the object store, as discussed in “More on Copydata” on page 195.

`otherOutputClass`: *class*

This optional keyword argument specifies the class of the object returned by the importer. The output class must be a subclass of `MoviePlayer`. This keyword argument allows you to import movies as an instance of a user-defined subclass of `MoviePlayer`, if desired.

This importer processes an input file in Apple’s QuickTime format or AVI format and returns a `MoviePlayer` or `InterleavedMoviePlayer` object. The importing process also creates a `DigitalAudioPlayer` instance and `DigitalVideoPlayer` instance to play the audio and video streams respectively, and synchronizes them to the movie player.The importing process returns the `MoviePlayer` or `InterleavedMoviePlayer object`.

If the data to be imported is in a “flattened” QuickTime file (that is, it contains no embedded Macintosh resources), or an AVI file, *source* must be given as a `Stream`. The importer opens the file to extract the movie data.

If the file is a QuickTime file containing embedded Macintosh resources, the *source* argument must be a `Collection` of strings that describe the exact pathname for the file, For now, the file path must always start from the root directory. For example, to specify a file path for a file called `movie1` stored in the `Cows` subdirectory of the `Animals` directory that resides on HD:

```
path := #("HD", "Animals", "Cows", "movie1")
```

If you want to save the `MoviePlayer` or `InterleavedMoviePlayer` to a title container, you must supply a container as the `container` keyword argument. This container is used to store a stream containing only the raw data for the media. You must additionally append the movie player, or any object that refers to it, to the title container when the importing process is finished.

The following code sample shows how to import and play an AVI or flattened QuickTime movie stored in the file “whale.mov” (assuming the file is in the same directory as the script file):

```
global whalestream := getstream theScriptDir "whale.mov" @readable

-- if whale.mov is a flattened QuickTime file:
global whaleplayer := importMedia theImportExportEngine
   whalestream @movie @quicktime @movieplayer

-- if whale.mov is an AVI file:
```

```
global whaleplayer := importMedia theImportExportEngine
   whalestream @movie @AVI @interleavedMovieplayer

global w:= new window boundary:(whaleplayer.bbox)
w.x := w.y := 40
show w
append w whaleplayer
play whaleplayer
```

The following code shows how to import the movie if `whale.mov` is a QuickTime file with embedded resources:

```
global path1 := theScriptDir as sequence
append path1 "whale.mov"
global whaleplayer := importMedia theImportExportEngine
   whalestream @movie @AVI @movieplayer
global whaleplayer := importMedia theImportExportEngine whalestream \
   @movie @quicktime @movieplayer
global w:= new window boundary:(whaleplayer.bbox)
w.x := w.y := 40
show w
append w whaleplayer
play whaleplayer
```

The following code shows how to save the imported movie to a title container.

```
global tc := new titleContainer path:"mytitle.sxt"

global whalestream := getstream theScriptDir "whale.mov" @readable
global whaleplayer := importMedia theImportExportEngine \
   whalestream @movie @quicktime \
   @InterleavedMoviePlayer container:tc
global w:= new window boundary:(whaleplayer.bbox)
w.x := w.y := 40
show w
append w whaleplayer
play whaleplayer

-- save the player to the title container
append tc whalePlayer

-- you also need to write a startup action
-- and save the title container
```

Note – If importing a movie generates an error message such as "couldn't find moov resource" it means you probably tried using a stream as the source file rather than a collection, or the other way around.

MoviePlayer versus InterleavedMoviePlayer

If you intend to play the imported movie from a CD-ROM drive in the future, you should import it as an `InterleavedMoviePlayer` to preserve the interleaving. Non-interleaved movies can play at a reasonable speed from a hard disk, but when played from a CD, both the audio and video might skip during playback.

When you import a movie file to an InterleavedMoviePlayer (rather than a MoviePlayer), the audio and video data are copied from the QuickTime file into a single ByteStream, preserving the interleaving as specified in the QuickTime file. At present, the importer only transfers data from the first video track and first audio track encountered.

When a frame of the movie plays, data is needed from both the video stream and the audio stream. If the streams are not interleaved, the video data needed for a frame may be arbitrarily distant on the disk from the audio data needed for the same frame. When playing back movies with non-interleaved data from a hard disk, the extra search time required to seek to non-sequential positions is relatively small and does not significantly affect the speed of playback. However, the search time becomes significant if the movie is played back from a CD.

When the audio and video streams are interleaved, the video data and audio data required for a frame are located sequentially on the disc, thus minimizing the search time between each frame.

Note – You are recommended to ALWAYS import your movies as interleaved movie players. In fact, if you try to import a movie as a MoviePlayer instead of an InterleavedMoviePlayer you might find the importing doesn't work, or it might return an interleaved movie player anyway.

More on Copydata

If the optional copydata keyword argument is given as false when importing a movie to an InterleavedMoviePlayer, the importing process does not copy the movie data into ScriptX. In this case, the player's interleavedStream instance variable points to a machine-specific byte stream. If you save the imported interleaved movie player to a title container you must take some additional steps, since a title is not portable across platforms if it contains a machine-specific stream.

Before adding the interleaved movie player to a title container, set its interleavedStream instance variable to undefined. Add the InterleavedMoviePlayer to a TitleContainer. Specify a startupAction for the TitleContainer that will get an appropriate stream (the original movie file) and put it in the interleavedStream instance variable of the interleaved movie player.

You may find it very useful to set the copydata argument to false when developing your scripts, and then set it to true (or don't supply it) when importing the movie with the intention of saving it to a title container.

Batch Processing

Multimedia titles typically contain hundreds of individual media data items. If you use external tools to create these items, it can be tedious to execute individual ScriptX commands to import each item. You can use batch processing to automate the conversion process so that it can be done without user intervention.

Batch processing involves creating a list of files, and then processing every element in the list. If you use an exception handler to catch any errors and to flag unprocessed files, the rest of the import process can continue uninterrupted.

The following code shows a routine that imports a set of PICT files as ScriptX bitmaps. The routine assumes that every file in the `/HD/Title/cv` directory is to be converted. It also assumes the existence of a `BitmapList` class which can hold a large number of bitmaps.

```
global pictdir := spawn theRootDir "\HD\Title\cv"
global fileCount := 0
global failCount := 0
global failList := new LinkedList
global images := new BitmapList
for pfile in (getContents pictdir) do (
    local strm
    if (isFile pfile) do (
        fileCount := fileCount + 1
        guard (
            strm := getStream pictdir pfile @readable
            bm := importMedia theImportExportEngine \
                    strm @image @pict @bitmap)
            append images bm
        ) catching
            all : (
                append failList pfile
                failCount := failCount + 1
                caught undefined
                )
            on exit
                plug strm
        end
        )
format debug "Processed %1 files, %2 failed\n" \
    #(fileCount, failCount) empty
```

At the end of the batch processing, a list of the files that were not converted is available in the `failList` object.

CHAPTER

16

Using the Director-to-ScorePlayer Importer

This chapter explains how to use the Director-to-ScorePlayer Importer to convert simple animations built in Macromedia™ Director® title to a ScriptX™ title.

This chapter does not teach Director—it assumes that if you're trying to import Director movies, you already know how to use Director.

The current release of the Director-to-ScorePlayer Importer runs only on Macintosh computers. Director titles contain Macintosh resource objects that cannot be read on other platforms. However, after a Director title has been imported into ScriptX, the resulting ScriptX animation can run on any platform that supports ScriptX.

The Director-to-ScorePlayer Importer converts a frame-based animation created in Macromedia Director to a hierarchy of objects in ScriptX. The score of a Director animation is reproduced in ScriptX by the `ScorePlayer` class, which supports a timeline. The `ScorePlayer` class is a subclass of `Player`, and has methods for playing, stopping, rewinding, changing the rate of the animation, and so on.

Before reading about the Director-to-ScorePlayer Importer, you may find it helpful to familiarize yourself with the ScriptX `Player` class, described in the *ScriptX Class Reference*.

After importing a Director animation into ScriptX you can modify the way the animation behaves. For example, you can change the positions of sprites in the animation, add new sound effects, and so on. See "Modifying an Imported Animation" on page 211 for details.

The Director-to-ScorePlayer Importer is built on the Director Translation Kit. For information about the Director Translation Kit API, see Chapter 19, "Director Translation Kit API."

What the Director-To-ScorePlayer Importer Can Import

A Macromedia Director title consists of a multitrack, linear stream of data. Each track can contain animation, video, graphics, audio, MIDI data, text, or a Lingo™ script. You can use the Director-to-ScorePlayer Importer to import

- a Macromedia Director castlist for either Director 3.1.3 or Director 4.0.
- a Macromedia Director score for either Director 3.1.3 or Director 4.0. The score consists of up to 24 or 48 channels of animation, 5 effects channels, and a script channel.

You can use the classes in the Director Translation Kit (DTK) to create custom importers that convert the individual components of Macromedia Director titles to ScriptX objects. For information about the DTK, see Appendix A.

What the Director-to-ScorePlayer Importer Cannot Import

This section describes the components of a Macromedia Director title that the Director-to-ScorePlayer Importer cannot import. The current version of the Director-to-ScorePlayer Importer cannot import:

- scores that use shared castmembers.

- Lingo scripts

  The current version of the Director-to-ScorePlayer Importer cannot convert Lingo scripts to ScriptX. However, the ScriptX language can be used to create methods or functions which perform operations equivalent to those programmed in Lingo. The import imports lingo scripts into ScriptX unconverted.

- film loops

  The current version of the Director-to-ScorePlayer Importer cannot import film loops into ScriptX. You can, however, write custom ScriptX commands that will force the `ScorePlayer` to repeat certain portions of the animation. See the discussion of Players in the *ScriptX Class Reference* for additional information.

- color ink effects

  The current version of the Director-to-ScorePlayer Importer can convert ink effects to ScriptX. However, it cannot convert color ink effects.

- palette transitions and cycling

  The current version of the Director-to-ScorePlayer Importer can import palettes into ScriptX. However, it cannot convert palette transitions or cycling effects.

- some types of transitions

  The current version of the Director-to-ScorePlayer Importer imports only wipe, barndoor, iris, and slide transitions. However, ScriptX directly implements many additional transitions.

- Wait for QuickTime Done tempo control.

  The current version of the Director-to-ScorePlayer Importer cannot import WaitForQuickTimeDone commands, but it can import WaitForSoundDone commands.

- XObjects

  The current version of the Director-to-ScorePlayer Importer cannot import XObjects, which is the MacroMedia specified extension to Lingo.

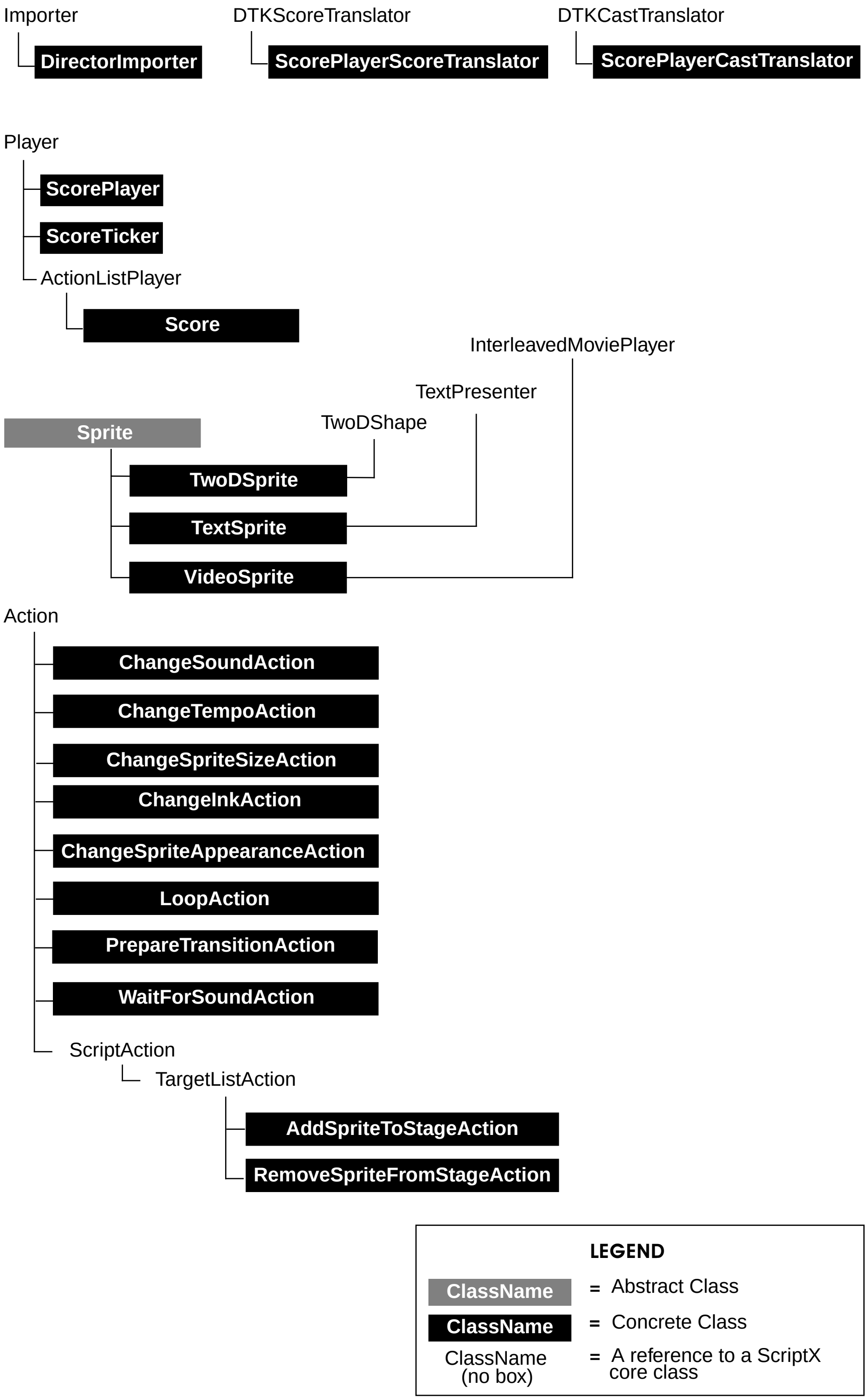
- extensions to HyperTalk®—such as XCMDs and XFCNs

  The current version of the Director-to-ScorePlayer Importer cannot convert Director HyperTalk extensions, such as XCMDs and XFCNs to ScriptX. However, the ScriptX core classes provide the functionality of many typical XCMDs and XFCNs.

- buttons—not supported
- 32-bit bitmap graphics—not supported

Classes and Inheritance

The following figure shows the inheritance hierarchy for the classes in the Director-to-ScorePlayer Importer API.



For detailed information about the classes in the Director-to-ScorePlayer Importer API, see Chapter 17, “Director-to-ScorePlayer Importer API” which lists these classes in alphabetical order.

DirectorImporter—imports a score and its castlist from Macromedia Director into ScriptX.

ScorePlayerScoreTranslator—a specialized subclass of the DTKScoreTranslator class that translates a Director score to a ScriptX ScorePlayer.

ScorePlayerCastTranslator—a specialized subclass of the DTKCastTranslator class that translates cast members in the Director cast list to TwoDSprite, TextSprite and VideoSprite instances.

ScorePlayer—a specialized subclass of the Player class that controls the playback of an imported Director score. For example, to play or stop the animation, call the play or stop methods on the ScorePlayer instance.

Score—a specialized subclass of the ActionListPlayer class that represents an imported Director score as a list of actions.

Sprite—provides functionality for presenters that represent castmembers in an imported Director score.

> TwoDSprite—represents a castmember from an imported Director score that displays a shape or a bitmap.
>
> TextSprite—represents a castmember from an imported Director score that displays text.
>
> VideoSprite—represents a video castmember from an an imported Director score.

SpriteChannelInfo—holds information about the state of a channel during an animation playback.

ChangeSoundAction—a specialized subclass of the Action class that represents sound changes in an imported Director score.

ChangeTempoAction—a specialized subclass of the Action class that represents tempo changes in an imported Director score.

AddSpriteToStageAction—a specialized subclass of the TargetListAction class that represents the appearance of a castmember in a previosuly empty animation channel during the playback of an imported Director score.

RemoveSpriteFromStageAction—a specialized subclass of the TargetListAction class that represents the disappearance of a castmember from an animation channel during the playback of an imported Director score.

ChangeSpriteSizeAction—a specialized subclass of the Action class that represents size changes in a sprite during the playback of an imported Director score.

ChangeInkAction—a specialized subclass of the Action class that represents ink mode changes during the playback of an imported Director score.

ChangeSpriteAppearanceAction—a specialized subclass of the Action class that represents changes in image or text during the playback of an imported Director score.

PrepareTransitionAction—a specialized subclass of the Action class that represents the start of a transition during the playback of an imported Director score.

ScoreTicker—a specialized subclass of the Player class that controls the timing for an imported Director score.

How Director Titles are Converted to ScriptX Objects

When importing a Director title into ScriptX as a ScorePlayer, you can import both the cast and the score, or either the cast or score.

Cast Translation

The Director-to-ScorePlayer Importer creates a ScorePlayerCastTranslator instance that translates each member of the castlist into an appropriate ScriptX object. The ScorePlayerCastTranslator class inherits from the class DTKCastTranslator in the Director Translation Kit component. See Chapter 18, "Using The Director Translation Kit" for information about cast translators in the DTK component.

Director castmembers are converted to ScriptX objects as follows:

- Director shape castmembers become TwoDSprite instances whose target instance variable contains a suitable Stencil instance.
- Director bitmap castmembers become TwoDSprite instances whose target instance variable contains a suitable Bitmap instance.
- Director text castmembers become TextSprite instances whose target instance variable contains a Text object.
- Director video castmembers become VideoSprite instances whose target instance variable contains DigitalVideoPlayer and DigitalAudioPlayer instances.
- Director sound castmembers become AudioStream objects.
- Director button castmembers are not translated.

Score Translation

The Director-to-ScorePlayer Importer creates a ScorePlayerScoreTranslator instance that translates a Director score to a ScriptX ScorePlayer. The ScorePlayerScoreTranslator inherits from the class DTKScoreTranslator in the Director Translation Kit component. See Chapter 18, "Using The Director Translation Kit" for information about score translators in the DTK component.

In a Director animation, a score consists of a set of frames. Each frame records what each castmember in each channel in the animation is doing in that frame. The Director-to-ScorePlayer Importer converts a Director score to a ScriptX ScorePlayer object. A ScorePlayer object does not have a frame-by-frame record for a score. Instead, it implements scores as a series of actions.

Before translating the score, the importing process creates an instance of Score, a subclass of ActionListPlayer that holds an action list representing an imported Director score.

The translateFrame method of the ScorePlayerScoreTranslator creates Action instances to represent changes in each frame, such as change of sprite, change of position, change of ink mode and so on. Each action is added to the action list in the Score instance.

For more information on the class ActionListPlayer, see "How an ActionList Player Works" on page 211.

Preparing a Director Title for Importing Into ScriptX

This section provides some guidelines for preparing a Macromedia Director title or component for importation into ScriptX.

- Ensure that the title to be imported does not have a shared cast, otherwise the importing process will not complete successfully. If the Director title does use a shared cast, use the facilities provided within the Director application to copy the cast to the score before saving the title.

- As far as possible, set up the animation so that each channel records the actions of a single "actor", even if it is made up of multiple castmembers. For example, if the animation contains a jogger, ensure that the jogger always appears in the same channel, even if it is made up of different bitmaps tha portray the jogger in different positions. This step is not essential to the importing process, but it will make it easier for you to modify the title if necessary after it has been imported into ScriptX.

Loading the Director-to-ScorePlayer Importer

The files needed for the Director-to-ScorePlayer Importer and the Director Translation Kit (which are used by the Director-to-ScorePlayer Importer) are contained in two folders called `dirimp` and `DTK`. Make sure that the `dirimp` and `DTK` folders are in the same folder as your ScriptX executable.

To load the Director-to-ScorePlayer Importer, open the title container saved to the file `dirimp.sxt` in the `dirimp` directory.

```
open TitleContainer path:"dirimp/dirimp.sxt"
```

You can also use the **Open** command from the File menu to open the title container.

When this file has finished loading, it leaves you in the `scratch` module. However, to use the Director-to-ScorePlayer Importer, you must work in the `DirectorImporter` module. To use an imported animation, you must work in the `ScorePlayer` module.

The best way to handle the module-switching is to create modules of your own that use the necessary modules, as discussed in "Saving an Imported Animation to a Title Container" on page 207.

Importing a Director Title as a ScorePlayer

To import a Director title into ScriptX you can do either of the following:

- Use a convenience function called `ImportDirector` which lets you interactively pick a file to import.
- Use the regular importing mechanism described in *ScriptX Developer's Guide*, (that is, call the `importMedia` method on `theImportExportEngine`).

Using the ImportDirector Function

The `importDirector` function opens up a file panel from which you can pick the Director file to import. The importing process returns a `ScorePlayer` object. This `ScorePlayer` object has an associated `Space` object stored in its `stage` instance variable. Before you can see the animation as it plays, you must append the `ScorePlayer` instance's space to a visible space, such as a `Window` object.

The `importDirector` function takes an optional `container` keyword that specifies a storage container in which to store the imported media for the animation. See "Saving an Imported Animation to a Title Container" on page 207 for an example of using the `importDirector` function with the `container` keyword.

The Director-to-ScorePlayer Importer uses modules. The classes to do the actual work of importing reside in a module called `DirectorImporter`. The classes created by the importing process to represent the animation reside in a module called `ScorePlayer`. You must work in the appropriate modules to use these classes.

The best way to handle the module-switching is to create modules of your own that use the necessary modules. If you are doing preliminary investigation and do not intend to save the imported animation to a title container then you can create a single module that uses both the `DirectorImporter` module and the `ScorePlayer` module and then work in that module. If you intend to save the imported animation to a title container, the module-switching is more complex, as discussed in "Saving an Imported Animation to a Title Container" on page 207.

Here is an example of how to import a Director title into ScriptX using the `ImportDirector` convenience function, and then play the imported animation. This example assumes that the imported animation will not be saved to a title container.

```
-- Load the Director-to-ScorePlayer importer
open TitleContainer path:"dirimp/dirimp.sxt"

-- Create a module that uses DirectorImporter and ScorePlayer modules

module SomeModule
   uses directorimporter
   uses scoreplayer
   uses scriptX
   exports scoreplayerA
end

-- Switch to working in the new module
in module SomeModule

-- Import the animation and return a score player.
-- An interactive file selection box appears on the screen so you
-- can select the Director file to import.
global scoreplayerA := ImportDirector()

-- create and show a window to present the animation
global w := new window
show w

-- append the scoreplayer's space to the window
append w scorePlayerA.stage

-- make sure the scoreplayer is at the beginning then play it
gotobegin scorePlayerA
play scorePlayerA
```

Note – You may find that your imported animation plays slowly the first time through. This is because the ink modes are being set on the fly. Subsequent play throughs will be faster, unless the ink modes change during the course of the animation.

Using the Import/Export Engine

The `ImportDirector` function provides a cover to using the Import/Export engine to import a Director title into ScriptX. If you prefer, you can use the Import/Export engine directly.

After the importer has been registered with ScriptX, you can call the `importMedia` method on the `theImportExportEngine` instance to import a Director title into ScriptX.

importMedia

importMedia theImportExportEngine *source dataCategory dataFormat outputType*
[`container`: *container*] ⇨ `ScorePlayer`

source — The source of the data to be imported.
For most importers, the *source* parameter may be a ScriptX `Stream` object or a `Collection` object identifying the path and filename. However, because the Director-to-ScorePlayer Importer needs access to individual resources, you must specify *source* as a collection rather than a stream. See "Specifying the Source File" on page 206 for an example.

dataCategory — The category of the imported data, which in this case is `@metaphor`.

dataFormat — The format of the imported data, which in this case is `@Director`.

outputType — The output type of the imported data, which in this case is either `@castList` or `@ScorePlayer`.
An output type of `@CastList` results in an `Array` object that contains the imported data. When the output is specified as @`CastList`, only the cast members are imported.
An output type of `@ScorePlayer` results in a `ScorePlayer` object being created to hold the animation. When the output type is `@ScorePlayer`, both the castlist and the score are imported.

container — The optional `container` argument specifies a storage container in which the converted elements will be stored.

When a Macromedia Director castmember list is imported into ScriptX, each castmember in the movie score or animation becomes an instance of either `TwoDSprite` or `TextSprite`, depending on whether it is an image (a shape or a bitmap) or a piece of text.

Note – The Director-to-ScorePlayer Importer cannot import a Macromedia Director movie score or animation that uses a *Shared Cast.* To import such a movie score or animation, you must first copy the shared castmembers from the *Shared Cast* movie to the movie score or animation.

When the importer imports a Director score into ScriptX, it creates and initializes a `ScorePlayer` instance, and creates a stage for it.

The stage is a `Space` or a `TwoDMultiPresenter` instance in which the animation appears when the `ScorePlayer` plays. The stage is the same size as the stage in Director.

Specifying the Source File

Since the importer needs access to individual resources, you must specify the *source* parameter for the `importMedia` method as a collection containing the directories and files that make up the pathname of the file containing the data to be imported.

To create a collection that specifies the full path, you can use one of two approaches. Either specify the path directly as a collection, or coerce a `DirRep` object to an `Array`, as the examples below show. In the first example, `myProject` is a directory that contains the file `filename.mm`.

```
src := (spawn theStartDir "myProject") as Array
```

```
append src “filename.mm”

src := #("HD", "Titles", "myOtherProject")
```

Saving an Imported Animation to a Title Container

The Director-to-ScorePlayer Importer uses modules. The classes that do importing reside in a module called `DirectorImporter`. The classes created by the importing process to represent the animation reside in a module called `ScorePlayer`. To import an animation and then use and save the imported animation, you must work in the appropriate modules.

The best way to handle the module-switching is to create modules of your own that use the necessary modules.

When you save an imported animation to a title container, you ideally want to save the objects and classes in the animation but you do not want to save the classes whose sole responsibility is to execute the importing process. Such classes include the `ScorePlayerScoreTranslator` and `ScorePlayerCastTranslator`. These classes reside in the `DirectorImporter` module.

We recommend that you work in a module that uses the `ScorePlayer`, `ScriptX` and `DirectorImporter` modules while importing the Director animation. When the importing process is finished, switch to a module that uses the `ScorePlayer` and `ScriptX` modules, (but not the `DirectorImporter` module). In this module, create the window to show the animation. Add this module to the title container before saving the title container. If you do this, the score player and all the other objects needed for the animation, along with the window to display the animation, will all be added to the title container. The objects used to execute the translating process will not be added to the container, since they are in a different module.

The steps for importing and saving a Director animation in ScriptX are summarized here:

1. Load the Director-to-ScorePlayer Importer.
2. Create a module that uses the `ScriptX` and `ScorePlayer` modules. This new module will be refferred to as *theTitleModule*.
3. Switch to working in *theTitleModule*.
4. Open a title container.
5. Create a module that uses the `ScriptX`, `ScorePlayer`, `DirectorImporter` and *theTitleModule* modules.
6. Switch to working in the new module.
7. Import the Director animation into ScriptX using either the `importDirector` function or the the `importMedia` method. In both cases, be sure to supply the storage container of the title container as the optional `container` keyword.
8. Switch back to working in *theTitleModule*.
9. Create and show a window, and then append the score player returned by the importing process to the window.
10. Define a `startAction` function for the title container. (Most likely you will want this function to call the `play` method on the score player to start the animation playing).
11. Add *theTitleModule* to the title container.

    This action adds all the objects in *theTitleModule* to the title container.
12. Close the title container.

Sample Script that Imports and Saves an Animation in ScriptX

This sample script illustrates how to create and switch between modules when importing a Director animation into ScriptX and saving the imported animation in a title container.

```
-- Load the Director-to-ScorePlayer Importer:

open TitleContainer path:"dirimp/dirimp.sxt"

-- Create a module that uses the ScriptX, Scratch and ScorePlayer
-- modules. This module exports the variable scoreplayerA so that later
-- you can use it to refer to the score player when the animation is
-- reloaded from the object store.

module MyTitleModule
    uses ScriptX
    uses Scratch
    uses ScorePlayer
    exports scoreplayerA
    exports tc
end

-- Switch to working in the MyTitleModule:

in module MyTitleModule

-- Define the globals used in this module:

    global scoreplayerA := undefined
    global tc := undefined
    global w

-- Create the title container to hold the imported animation.
-- This title container will reside in the module MyTitleModule:

    global tc := new TitleContainer path:"myTitle.sxt"

-- Define a transient module that references the DirectorImporter
-- module which contains classes used to do the importing.

module temp
    uses ScriptX
    uses DirectorImporter
    uses ScorePlayer
    uses MyTitleModule
end

-- Switch to working in the temp module.

in module temp
    -- Import the Director movie
    scoreplayerA := ImportDirector container:tc

-- Switch back to working in MyTitleModule
in module MyTitleModule

-- Create a ScriptX window:
    w := new Window bbox:(ScorePlayerA.stage.boundary)
```

```
-- Append the score player's space to the window:
   append w scoreplayerA.stage

-- Show the window:
   show w

-- Add everything in MyTitleModule to the storage container:
   append tc (getModule @MyTitleModule)

-- Define the startup function for the title container.
-- Note that scorePlayerA has been exported as a global
-- variable for this module.

tc.startupAction := (self ->
         (
            gotobegin scoreplayerA
            play scorePlayerA
         )
      )

-- Close the storage container to save the animation
close tc
```

Your imported animation is now saved as a title container called "myTitle.sxt" in the ScriptX startup directory.

Reloading the Saved Animation

After quitting from ScriptX, you can run the animation in any of the following ways:

- Double click on the "myTitle.sxt" icon.
- Drag and drop the "myTitle.sxt" icon on to the ScriptX application icon.
- Open ScriptX from the ScriptX application icon, then open the file "myTitle.sxt."

When the myTitle.sxt title opens, it runs through the animation once. After making sure you are in the correct module (MyTitleModule) you can use the variable ScorePlayerA to refer to the score player that controls the animation.

```
in module mytitlemodule

-- rewind the animation to the beginning
gotobegin scorePlayerA

-- play the animation again
play scorePlayerA

-- stop the animation
stop scorePlayerA

-- change the width of the window presenting the animation.
-- The window is pointed to by the scoreplayer’s stage’s
-- presentedBy instance variable.
scorePlayerA.stage.presentedby.width := 300
```

Playing an Imported Title

After importing a Director animation into ScriptX as a ScorePlayer object, or after opening a title container that contains a saved animation, you can run and stop the animation by calling methods on the score player that represents the animation.

The class ScorePlayer inherits from the class Player. It supports most of the usual Player methods, such as play, pause, stop, gotobegin, gotoend and fastForward. The method playUntil does not work for score players. See the *ScriptX Class Reference* for documentation on all the Player methods.

Score players can only play forwards. They cannot play backwards.

To present an animation in a window, append the score player's stage to the window:

```
w := new window
show w
append w scorePlayerA.stage
```

To play an animation, call the play method on the score player:

```
play scorePlayerA
```

To stop an animation, call the stop method on the score player:

```
stop scorePlayerA
```

To move an animation back to the beginning status, call the gotobegin method on the score player:

```
gotobegin scorePlayerA
```

To move an animation to any particular frame, set the frame instance variable on the ScorePlayer. For example, to move it to frame 8:

```
scorePlayerA.frame := 8
```

The frame instance variable is local to the class ScorePlayer, it is not inherited from Player. For most Player classes, you move the player to a particular point in its media by setting the time instance variable, but for the ScorePlayer class, set the frame instance variable instead.

To step an imported animation to its next frame, increment the frame instance variable of a ScorePlayer by 1.

```
scoreplayerA.frame := scorePlayerA.frame + 1
```

To set an animation into looping mode, so that when it reaches the end it starts over again automatically, set the looping instance variable of the score player to true.

Changing the Rate of the Animation

A score player has a scoreTicker instance variable, whose value is a ScoreTicker instance. A score ticker is basically a clock that has a callback that fires every tick to step the animation through one frame. .

The rate of the score player effects the rate of play of the animation since it is a master clock of the score ticker. (The rate of the `Score` instance does not effect the rate of the animation. The rate of the `Score` instance is always 0.) For more information on how master clocks affect the rate of their slave clocks, see the "Clocks" chapter in the *ScriptX Components Guide*.

To double the speed of an animation whose `ScorePlayer` has a current rate of 1:

```
scorePlayerA.rate := 2
```

The animation tries to play at the required speed. If the rate is too fast, the animation does not skip frames but plays as fast as it can. The disadvantage to this approach is that the movement of images on the screen may not stay in synch with the sound for the animation.

Modifying an Imported Animation

After importing a Director score into ScriptX, you can modify the objects that implement the animation in ScriptX. To modify the steps of an animation, add or modify the actions in the action list for the animation. The action list is stored in the `actionList` instance variable of the `Score` object associated with the the `ScorePlayer`.

How an ActionList Player Works

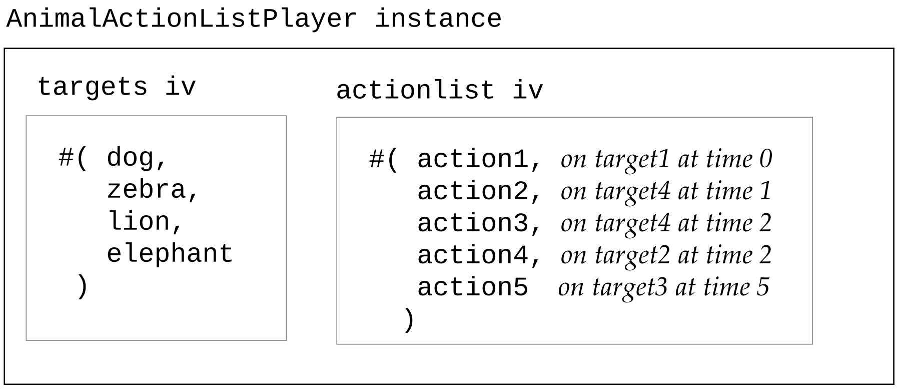
Understanding how the class `ActionListPlayer` works will help you understand how the class `Score` works, since `Score` inherits much of its functionality from `ActionListPlayer`. This section gives a quick description of how the class `ActionListPlayer` works. For more information on this class, see the "Animation" chapter of the *ScriptX Components Guide*.

An `ActionListPlayer` can be used to play a series of actions. It has two lists:

- a list of actions stored in its `actionList` instance variable. Each action is itself an object, an instance of one of the classes of `Action`.
- a list of targets stored in its `targets` instance variable. Each target is an object.

Each action has an instance variable `targetNum` that tells it which number target it affects in the target list. It also has a `time` instance variable that tells it what time to activate.

For example, consider the hypothetical object `AnimalActionListPlayer`. The value of its `targets` instance variable is the array `#(dog, zebra, lion, elephant)`. The value of its `actionlist` instance variable is the array `#(action1, action2, action3, action4, action5)`, as illustrated in Figure 16-1, "The AnimalActionListPlayer instance".



```
AnimalActionListPlayer instance

targets iv          actionlist iv

#( dog,             #( action1, on target1 at time 0
   zebra,              action2, on target4 at time 1
   lion,               action3, on target4 at time 2
   elephant            action4, on target2 at time 2
 )                     action5  on target3 at time 5
                     )
```

Figure 16-1: The AnimalActionListPlayer instance

When the `animalActionListPlayer` is played by the command:

```
play AnimalActionListPlayer
```

the following happens:

`action1` is performed on the `dog` instance at time 0

`action2` is performed on the `elephant` instance at time 1

`action3` is performed on the `elephant` instance at time 2

`action4` is performed on the `zebra` instance at time 2

`action5` is performed on the `lion` instance at time 5

What is the Target of An Action in an Animation?

When creating a new action, you specify the time at which the action is to occur, along with a `targetNum` value that points to the object affected by the action. The object in the `targetNum` position of the collection in the `targets` instance variable of the appropriate `ActionList` instance is the object to be affected by the action. In the following discussion, the term "the `targets` collection" means the collection in the `targets` instance variable of the `Score` object.

The tricky part when modifying the action list of a `Score` object is that the actions operate on items in the `targets` collection but the `targets` collection could change from frame to frame.

Each position in the `targets` collection corresponds to a channel in the original Director score. At any given time, the object in the *n*th position in the `targets` collection corresponds to the castmember that appears in the *n*th channel in the Director score **at that time**.

The `targets` collection updates as the animation runs so that it records which objects are playing in which channels at which time.

For example, suppose object A plays in channel 5 at time 10, but at time 11 object B appears in that channel instead. In this case, object A is in the fifth position in the `targets` collection at time 10, and object B is in the fifth position at time 11.

Handy Hints for Keeping Control of Targets

One way to improve the ease of modification of an Director title imported into ScriptX is to ensure that each channel in the Director title contains only one “actor” throughout the running of the title. The actor can be made up of multiple images, for example, a jogger might be made up of several different bitmaps portraying the jogger in different positions.

Having only one actor per channel in Director will help you know exactly which actor is in what position of the `targets` collection after the title is imported into ScriptX. For example, suppose the only actor to appear in channel 3 in a Director score is a jogger. After importing the title into ScriptX you can be sure that any action to be performed on the jogger must have a `targetNum` of 3.

The Classes of Action

To modify the actions in an imported animation in ScriptX, you add or modify actions in the action list. This section lists the possible actions in an animation, and the following sections help you read and modify the action list of an animation.

Most actions have `new` methods that take a `targetNum` keyword, a `time` keyword, and also action-specific keywords. In this context, the `targetNum` keyword specifies what channel of the animation the action will happen in and the `time` keyword specifies at what time the action will occur.

After creating a new action, add it to the action list of the appropriate `Score` object.

An action list is sorted by time. When you call the `append` or `prepend` method on an action list player, the newly added action is put in the correct place in the list according to its time. If several actions in the list have the same time, let’s say 10, then appending another action for time 10 causes the newly added action to appear after the other actions for time 10. Prepending another action for time 10 causes the newly added action to come first of all the actions for time 10.

Subclasses of Action

The subclasses of `Action` that are appropriate for use in an imported animation are:

DeltaPathAction

Moves an object by a relative amount from its current position. Its `deltaPosition` instance variable specifies the relative distance to move. For example, to create an action that moves the first object in the target list 10 pixels to the right and 20 pixels up from its current position:

```
actionx := new DeltaPathAction targetNum:1 time:10 \
deltaPosition:(new Point x:10 y:20)
-- append the new action to the actionlist
-- myScorePlayer is an existing ScorePlayer
myActionList := myscoreplayer.score.ActionList
append myActionList actionx
```

InterpolateAction

Moves an object in a smooth way to a given position at a given time. Its `destPosition` instance variable specifies the destination position, and its `destTime` instance variable specifies the time to arrive at the destination. For example, to create an action that moves the second object in the target list to the position (300, 500) in its space, to arrive there at time 20:

```
actionx := new InterpolateAction targetNum:2 time:15 \
destPosition:(new Point x:300 y:500) destTime: 20
myActionList := myscoreplayer.score.ActionList
append myActionList actionx
```

PathAction

Moves an object to an absolute position immediately. Its `destPostion` instance variable specifies the destination position. For example, to create an action that moves the fourth object in the target list to the position (100, 200) in its space:

```
actionx:= new PathAction targetNum:4 time:20 \
destPosition:(new Point x:100 y:200)
myActionList := myscoreplayer.score.ActionList
append myActionList actionx
```

ScriptAction

Performs an action described by a function, which allows you to do any kind of action you want. Its `script` instance variable contains the function to run. For example, to call the function `strobefn` on the fifth object in the target list:

```
actionx:= new ScriptAction targetNum:5 time:25 script:strobefn
myActionList := myscoreplayer.score.ActionList
append myActionList actionx
```

Note: The function `strobefn` takes 3 arguments—action, target and player, where player in this case is `myscoreplayer.score`.

TargetListAction

Executes a function and puts the object returned by the function into the specified position in the target list. This enables you to change the objects in a target list as an animation progresses. For example, to run the `makeNewBearFn` which makes a new `Bear` object, and installs it in place of the existing object at the seventh position in the target list:

```
actionx:= new TargetListAction targetNum:7 time:35 \
            script:makeNewBearFn
myActionList := myscoreplayer.score.ActionList
append myActionList actionx
```

For this to remove objects when it rewinds, you need to supply a rewind script for the target list action, which must clear out both the `targets` array and the `rewindScripts` array.

Actions Specific to Imported Animations

The `ScoreActionList` subclass of `ActionList` additionally supports the following classes of actions, which are created when the Director-to-ScorePlayer Importer is loaded.

ChangeTempoAction

Changes the tempo of a score. Its `tempo` instance variable contains the new tempo. Note that this action does not take a target object since it works on the score as a whole, not on any individual sprite in the animation. For example, to set the tempo to 15 at time 3, and then speed it up even more to 20 at time 8:

```
t1:= new ChangeTempoAction time:3 tempo:15
t2:= new ChangeTempoAction time:8 tempo:20
myActionList := myscoreplayer.score.ActionList
append myActionList t1
append myActionList t2
```

ChangeSoundAction

A `ChangeSoundAction` tells one of the two `DigitalAudioPlayer` instances to start playing a sound. It has `time`, `mediaStream` and `audioPlayerIndex` instance variables, as follows:

`time` Time for the sound to start

`mediaStream`

A `MediaStream` object that contains digitized sound. See the previous chapter to see how to use the Import/Export engine to import a sound file into ScriptX as a `mediaStream` object.

`audioPlayerIndex`

An integer (which must be 1 or 2) that represents which channel should play the sound.

So for example to play the sound in the media stream `LionRoaring` at time 100 on channnel 2:

```
roar1 := new ChangeSoundAction time:100 mediaStream:LionRoaring \
   audioPlayerIndex:2
myActionList := myscoreplayer.score.ActionList
append myActionList roar1
```

RemoveSpriteFromStageAction

A `RemoveSpriteFromStageActionAction` removes the sprite in the given channel from that channel.

For example, to make the sprite in channel 2 disappear at time 15:

```
spriteDisappear1 := new RemoveSpriteFromStageAction targetNum:1 \
   time:15
myActionList := myscoreplayer.score.ActionList
append myActionList spriteDisappear1
```

AddSpriteToStageAction

This action makes a sprite appear in the given channel at the given time. During the importing of a Director title into ScriptX, the Director-to-ScorePlayer Importer automatically creates an `AddSpriteToStageAction` action for each sprite in the animation to make it appear the first time. Note that to change the shape of a sprite in the animation, you should use the `ChangeSpriteAppearanceAction`.

`AddSpriteToStageAction` instances have a `target` instance variable that indicates which presenter should appear. The presenter must be either a `TwoDSprite` or a `TextSprite`.

If this is the first time that an image is to appear in this channel, the value of the boundary of this presenter is subsequently set by a `ChangeSpriteAppearanceAction`.

Note that three actions are required to make a sprite appear in a previously empty channel:

– An `AddSpriteToStageAction` puts a presenter in a channel that was previously empty.

– A `ChangeSpriteAppearanceAction` sets the target of the presenter. The value of the `target` keyword must be another presenter whose boundary instance variable points to the desired bitmap, shape or text.

– A `PathAction` sets the position of the presenter.

ChangeSpriteAppearanceAction

This action changes the image of the sprite in the given channel at the given time. When creating one of these actions, supply the `target` keyword, whose value must be a presenter, specifically, a `TwoDSprite` or a `TextSprite`.

This action does not actually put the presenter specified by `target` into the animation, instead it puts the object in the target's `boundary` instance variable into the animation. The action switches the value of the `boundary` instance variable of the sprite in the given channel to be the same as the `boundary` of the given target.

ChangeSpriteSizeAction

This action changes the width and or height of the sprite in the given channel at the given time. It takes `width` and `height` keywords.

To change the width and height of the sprite in channel 8 at time 9:

```
myActionList := myscoreplayer.score.ActionList
changeSize1 := new ChangeSpriteSizeAction targetNum:8 time:9 \
   width:100 height:120
append myActionList changeSize1
```

ChangeInkAction

This action changes the inkmode of the given channel at the given time. It takes an `inkmode` keyword, whose value can be `@copy, @matte` or `@invisible`.

`@copy` – When a sprite has an inkmode of `@copy`, the whole of the rectangle enclosing the image obscures any underlying images.

`@matte` – If the inkmode of an image is `@matte`, only image within the actual boundary of the shape obscure any underlying images.

`@invisible` – If the inkmode of an image is `@invisible`, only the colored pixels of the image obscure any underlying images, the white pixels are transparent.

These three inkmodes are illustrated in Figure 16-2. Other inkmodes default to `@copy`.

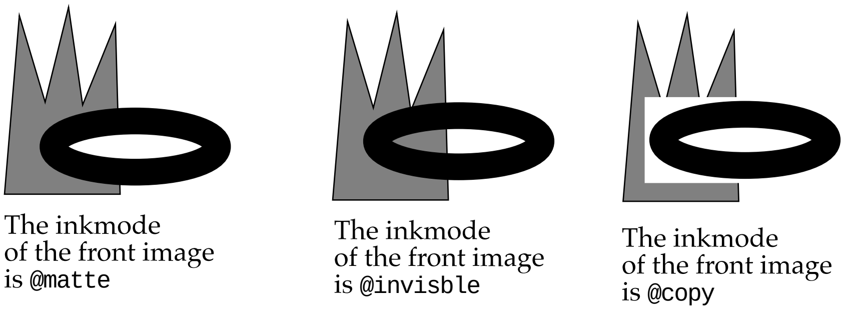


Figure 16-2: The @matte, @invisble and @copy inkmodes

To change the inkmode of channel 5 to `@matte` at time 77:

```
myActionList := myscoreplayer.score.ActionList
changeink1 := new ChangeInkAction targetNum:5 time:77 \
   inkmode:@matte
append myActionList changeink1
```

WaitForSoundAction

This action causes the animation to wait until a sound has finished. It has a `channelNumber` instance variable, whose value is 1 or 2, which indicates the sound channel to wait for. The value of the time instance variable indicates at what time to start waiting (that is, the animation continues playing until that time and then starts waiting).

To make the animation start waiting at time 10 for the sound in sound channel 1 to finish:

```
myActionList := myscoreplayer.score.ActionList
waitSound1 := new WaitForSoundAction channelNumber:1 time:10 \
append myActionList waitSound1
```

PrepareTransitionAction

This action sets up a transition.

When the importing process encounters a transition in a Director score, it creates two actions:

- A `PrepareTransitionAction` that prepares the score player for the transition.
- A `ScriptAction` that starts the transition playing. The value of the `script` instance variable is a global, exported function called `scorePlayerStartTransition`.

You can modify the `duration`, `direction` and `scale` instance variables of existing `PrepareTransistionAction` instances but you cannot add new `PrepareTransitionAction` instances

How to Read the Action List of a Score

To modify an animation that was created in Director and then imported into ScriptX, you add actions to, or modify the existing actions in, the action list of the `Score` object.

The value of the actionList instance variable of the Score object is a collection of actions. If you print the actionList you see a list of actions, for example:

```
-- myScorePlayer is a an existing scorePlayer
myactionlist := myScorePlayer.score.actionlist
```

returns the following action list:

```
#(ChangeTempoAction@0xad2d28, AddSpriteToStageAction@0xa06d48,
PathAction@0xad3a48, ScriptAction@0xad3d08,
AddSpriteToStageAction@0xa0aa08, PathAction@0xad3e28,
ScriptAction@0xad4048, AddSpriteToStageAction@0xa0aac8,
PathAction@0xad4228, ScriptAction@0xad4608, ...) as ActionList
```

While this collection shows the actions that make up the animation, you cannot see what objects perform the actions or at what times.

You can define a function that displays the actions in the action list with more information. The example function defined here uses the format method which prints a string, and substitutes values into variables in the string. (See Chapter 3 in the *ScriptX Language Guide* for more information on using the format method.)

```
function printActions x ->(
   format debug "%1 at time %2 on target number %3 \n" \
   #(x, x.time, x.targetnum) \
   #(@unadorned, @unadorned, @unadorned))

-- %1, %2, %3 refer to the items at postion 1, 2 and 3
-- in the input list, which is #(x, x.time, x.targetnum)
-- #(@unadorned, @unadorned, @unadorned) are the printing styles for
-- %1, %2 and %3 (in this case, print them all in unadorned style)
```

You can use the method forEach to call a function such as printActions on each item in a collection. The forEach method takes three arguments: the collection to iterate over, the function to call for each item in the collection, and another argument that can be used to pass additional information to the function. In this case, no additional information is needed, so you can pass anything as the third argument since it is ignored anyway. (See the description of the class Collection in the *ScriptX Class Reference* for more information about the forEach method.)

```
-- myScorePlayer is an existing scorePlayer
myactionlist := myScorePlayer.score.actionlist
forEach myactionlist printActions 1
```

This might return:

```
ChangeTempoAction@0xedb0a8 at time 0 on target number 1
AddSpriteToStageAction@0xe76548 at time 0 on target number 1
ChangeSpriteAppearanceAction@0xedb1a8 at time 0 on target number 1
PathAction@0xedb1e8 at time 0 on target number 1
AddSpriteToStageAction@0xe2ce88 at time 0 on target number 4
ChangeSpriteAppearanceAction@0xede6e8 at time 0 on target number 4
PathAction@0xede728 at time 0 on target number 4
AddSpriteToStageAction@0xe2cf08 at time 1 on target number 3
ChangeSpriteAppearanceAction@0xedee68 at time 1 on target number 3
PathAction@0xedeee8 at time 1 on target number 3
ChangeSpriteAppearanceAction@0xedef08 at time 1 on target number 4
PathAction@0xedfa48 at time 1 on target number 4
RemoveSpriteFromStageAction@0xdfefa8 at time 2 on target number 1
ChangeSpriteAppearanceAction@0xedfc08 at time 2 on target number 3
PathAction@0xedfce8 at time 2 on target number 3
ChangeSpriteAppearanceAction@0xedfa08 at time 2 on target number 4
```

```
PathAction@0xedfcb8 at time 2 on target number 4
ChangeSoundAction@0xedfab6 at time 2 on target number 0
ChangeInkAction@0xede688 at time 3 on target number 4
and so on...
```

This output shows what actions happen at what times. To help you understand the output, here's a line by line explanation.

At time 0, set the tempo of the animation. Note that the `targetNum` of a `ChangeTempoAction` is irrelevant since it operates on the animation as a whole, not on any particular object in the animation.

```
ChangeTempoAction@0xedb0a8 at time 0 on target number 1
```

At time 0, make a sprite appear in channel 1. To make an object appear the first time, a `AddSpriteToStageAction` action puts a sprite into the channel; a `ChangeSpriteAppearanceAction` gives the sprite an image, and a `PathAction` sets the position of the sprite. The position is determined by the path action's `destPosition` instance variable.

```
AddSpriteToStageAction@0xe76548 at time 0 on target number 1
ChangeSpriteAppearanceAction@0xedb1a8 at time 0 on target number 1
PathAction@0xedb1e8 at time 0 on target number 1
```

At time 0, make the sprite in channel 4 appear.

```
AddSpriteToStageAction@0xe2ce88 at time 0 on target number 4
ChangeSpriteAppearanceAction@0xede6e8 at time 0 on target number 4
PathAction@0xede728 at time 0 on target number 4
```

At time 1, make the sprite in channel 3 appear.

```
AddSpriteToStageAction@0xe2cf08 at time 1 on target number 3
ChangeSpriteAppearanceAction@0xedee68 at time 1 on target number 3
PathAction@0xedeee8 at time 1 on target number 3
```

At time 1, change the image of the sprite in channel 4 and move it. The destination position is determined by the `destPosition` on the `PathAction` instance.

```
ChangeSpriteAppearanceAction@0xedef08 at time 1 on target number 4
PathAction@0xedfa48 at time 1 on target number 4
```

At time 2, make the sprite in channel 1 disappear.

```
RemoveSpriteFromStageAction@0xdfefa8 at time 2 on target number 1
```

At time 2, change the image of the sprite in channel 3 and move it.

```
ChangeSpriteAppearanceAction@0xedfc08 at time 2 on target number 3
PathAction@0xedfce8 at time 2 on target number 3
```

At time 2, change the image of the sprite in channel 4 and move it.

```
ChangeSpriteAppearanceAction@0xedfa08 at time 2 on target number 4
PathAction@0xedfcb8 at time 2 on target number 4
```

At time 2, start playing a sound.

```
ChangeSoundAction@0xedfab6 at time 2 on target number 0
```

At time 3 change the ink mode of the sprite in channel 4

```
    ChangeInkAction@0xede688 at time 3 on target number 4
```

Retrieving Information about an Animation

You can use methods that operate on collections, such as `chooseOne` and `chooseAll` to answer questions such as:

- Is there a path action for a given channel at a given time?
- What are all the sprite appear actions?
- What are all the actions that happen at time 10?
- What `targetNum` does the first path action at time 5 operate on?
- and so on...

The following sample code uses the function `printActions` introduced earlier, which prints an action, its time and its target number. See page 218 for a definition of this function.

```
-- myScorePlayer is an existing scoreplayer
global myactionlist := myScorePlayer.score.actionlist

-- To print all the pathactions:
local pathActionList := chooseAll myactionlist \
    (a b -> (getClass a) = PathAction) \
foreach pathActionList printActions 1

-- print all the pathactions at time 8:
local pathActionListSub1 := chooseAll pathActionList \
    (a b -> a.time = 8) 1
foreach pathActionListSub1 printActions 1

-- to print all actions at time 8:
foreach (chooseAll myActionList (a b -> a.time = 8) 1) \
    printActions 1

-- to print the all the actions at time = 11 on targetNum 4:
foreach (chooseAll myActionList (a b ->
        (a.time = 11) and (a.targetNum = 4)) 1) \
        printActions 1
```

CHAPTER

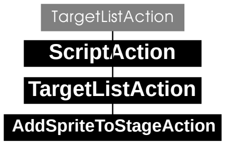
17 Director-to-ScorePlayer Importer API

This chapter lists the classes in the Director to ScorePlayer Importer component.

The classes in the Director Translation Kit live in the DTK module, so you must work in the `DirectorImporter` module, or in a module that uses the `DirectorImporter` module, to be able to use them.

For information about how to use these classes to import a Director title into ScriptX, see Chapter 16, “Using the Director-to- ScorePlayer Importer.”

AddSpriteToStageAction



Class type: Tool class (concrete)
Resides in: `dirimp.sxl`. Works in the KMP and the ScriptX executable.
Inherits from: `TargetListAction`

The `AddSpriteToStageAction` class is a specialized subclass of the `TargetListAction` class that represents the addition of a castmember to a channel. When importing a Macromedia Director score, the importer creates an instance of the `AddSpriteToStageAction` class whenever a castmember is added to a channel that was previously empty.

You can add `AddSpriteToStageAction` instances to the action list of a `Score` to add a castmember to a channel that was previously empty, or to bring back a castmember that was previously made invisible by a `RemoveSpriteFromStageAction` action.

Creating New AddSpriteToStageAction Instances

To create a new instance of the `AddSpriteToStageAction` class, call the `new` method, which takes `targetNum`, `time` and `target` keywords.

time The time at which the action is to occur.
targetNum The channel which the action affects.
target A `TwoDSprite`, `TextSprite` or `VideoSprite` object, depending on whether an image, text or movie is to be displayed.
If the sprite was previously displayed and was removed by a `RemoveSpriteFromStageAction` action, the `target` keyword must be the value in the `targetNum` position of the appropriate `Score` object's `targets` instance variable.
For example, if the value of `targetNum` is 3, and the `Score` is `actionplayer1`:

```
target: actionplayer.targets[3]
```

If this is the first time that an image is to appear in this channel, the value of the boundary of this presenter must subsequently be set by a `ChangeSpriteAppearanceAction`.

For example, to make make the character in channel 2 disappear at time 5 and then reappear again at time 15:

```
-- myscoreplayer is an existing ScorePlayer instance
myActionList := myscoreplayer.score.ActionList
spriteDisappear1 := new RemoveSpriteFromStageAction targetNum:2 \
   time:5
prepend myActionList spriteDisappear1
spriteAppear1 := new AddSpriteToStageAction targetNum:2 time:15 \
   target: (myscoreplayer.score.targets[2])
prepend myActionList spriteAppear1
```

Instance Variables

Inherited from `Action`:

```
authorData          targetNum          time
playOnly
```

Inherited from `ScriptAction`:

```
script
```

Inherited from `TargetListAction`:
    `rewindScript`

The following instance variables are defined in `AddSpriteToStageAction`:

target

self.`target` (read-write) `Number`

The sprite to to be added to a channel by the action *self*.

Instance Methods

Inherited from `Action`:
    `trigger`

ChangeInkAction

Class type: Tool class (concrete)
Resides in: `dirimp.sxl`. Works in the KMP and the ScriptX executable.
Inherits from: `Action`

The `ChangeInkAction` class is a specialized subclass of the `Action` class that represents a change in ink mode for a channel. When importing a Macromedia Director score, the importer creates an instance of the `ChangeSpriteAppearanceAction` class whenever a channel's ink mode changes.

You can add `ChangeInkAction` instances to the action list of a `Score` to make a channel change its inkmode.

Currenly, the only inkmodes supported are `@copy`, `@matte` and `@invisible`.

Creating New ChangeInkAction Instances

To create a new instance of the `ChangeInkAction` class, call the `new` method, which takes `targetNum`, `time`, and `inkmode` keywords.

time The time at which the action is to occur.
targetNum The channel which the action affects.
inkmode The new inkmode, which can be `@matte, @copy` or `@invisible`.

For example, to change the inkmode of the character in channel 5 to `@matte` at time 77:

```
-- myscoreplayer is an existing ScorePlayer instance
myActionList := myscoreplayer.score.ActionList
changeink1 := new ChangeInkAction targetNum:5 time:77 \
   inkmode:@matte
append myActionList changeink1
```

Instance Variables

Inherited from `Action`:

`authorData`	`targetNum`	`time`
`playOnly`		

The following instance variables are defined in `ChangeInkAction`:

inkMode

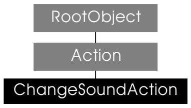
self`.inkMode` (read-write) `Number`

The ink mode to be set by the action *self* on the channel determined by the value in *self*'s `targetNum` instance variable. The ink mode can be `@matte`, `@copy` or `@invisible`.

Instance Methods

Inherited from `Action`:
`trigger`

ChangeSoundAction



Class type: Tool class (concrete)
Resides in: `dirimp.sxl`. Works in the KMP and the ScriptX executable.
Inherits from: `Action`

The `ChangeSoundAction` class is a specialized subclass of the `Action` class that represents sound changes. While the Director Importer is importing a Macromedia Director movie score or animation, each sound change (such as a sound starting to play) becomes a `ChangeSoundAction`. A `ChangeSoundAction` tells one of the two `DigitalAudioPlayer` instances to start playing a sound.

You can add `ChangeSoundAction` instances to the action list of a `Score` to add additional sound changes to an animation.

Creating New ChangeSoundAction Instances

To create a new instance of the `ChangeSoundAction` class, call the `new` method, which takes `time`, `mediaStream` and `audioPlayerIndex` keywords.

time Time for the sound to start

mediaStream

A `MediaStream` object that has sound in it. See the previous chapter for information on how to use the Import/Export engine to import a sound file into ScriptX as a `mediaStream` object.

audioPlayerIndex

An integer (which must be 1 or 2) that represents which channel should play the sound.

For example to play the sound in the media object `LionRoaring` at time 100 on channnel 2:

```
roar1 := new ChangeSoundAction time:100 mediaStream:LionRoaring \
   audioPlayerIndex:2
-- myScorePlayer is an existing ScorePlayer
myActionList := myscoreplayer.score.ActionList
append myActionList roar1
```

Instance Variables

Inherited from `Action`:

`authorData`
`playOnly`
`targetNum`
`time`

The following instance variables are defined in `ChangeSoundAction`:

audioPlayerIndex

self`.audioPlayerIndex` (read-write) `Number`

The value of this instance variable is either 1 or 2, indicating which sound channel to use to play a sound.

mediaStream

self`.mediaStream` (read-write) `MediaStream`

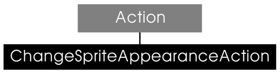
An `AudioStream` object that is associated with a `DigitalAudioPlayer` object when the `Score` triggers a `ChangeSoundAction`.

Instance Methods

Inherited from `Action`:

```
trigger
```

ChangeSpriteAppearanceAction



Class type: Tool class (concrete)
Resides in: `dirimp.sxl`. Works in the KMP and the ScriptX executable.
Inherits from: `Action`

The `ChangeSpriteAppearanceAction` class is a specialized subclass of the `Action` class that represents a change in appearance of a castmember. When importing a Macromedia Director score, the importer creates an instance of the `ChangeSpriteAppearanceAction` class whenever a castmember changes its appearance.

You can add `ChangeSpriteAppearanceAction` instances to the action list of a `Score` to make a castmember change its appearance.

Creating New ChangeSpriteAppearanceAction Instances

To create a new instance of the `ChangeSpriteAppearanceAction` class, call the `new` method, which takes `targetNum`, `time` and `target` keywords.

time The time at which the action is to occur.

targetNum The channel which the action affects.

target The `TwoDSprite` or `TextSprite` object whose `boundary` instance variable points to the desired image or text.

This action does not actually put the presenter specified by `target` into the animation, instead it puts the object in the target's `boundary` instance variable into the animation. The action switches the value of the `boundary` instance variable of the object in the given channel to be the same as the `boundary` of the given target.

Instance Variables

Inherited from `Action`:

```
authorData        targetNum        time
playOnly
```

The following instance variables are defined in `ChangeSpriteAppearanceAction`:

target

self`.target` (read-write) `Sprite`

The sprite whose appearance is to be changed by the action *self*.

Instance Methods

Inherited from `Action`:

```
trigger
```

ChangeSpriteSizeAction

Action
ChangeSpriteSizeAction

Class type: Tool class (concrete)
Resides in: `dirimp.sxl`. Works in the KMP and the ScriptX executable.
Inherits from: `Action`

The `ChangeSpriteSizeAction` class is a specialized subclass of the `Action` class that represents a change in size of a castmember. When importing a Macromedia Director score, the importer creates an instance of the `ChangeSpriteSizeAction` class whenever a castmember changes its size.

You can add `ChangeSpriteSizeAction` instances to the action list of a `Score` to make a castmember change its size.

Creating New ChangeSpriteSizeAction Instances

To create a new instance of the `ChangeSpriteAppearanceAction` class, call the `new` method, which takes `targetNum`, `time`, `width` and `height` keywords.

time The time at which the action is to occur.
targetNum The channel which the action affects.
width The new width of the castmember.
height The new height of castmember.

To change the width and height of the character in channel 8 at time 9:

```
-- myscoreplayer is an existing ScorePlayer instance
myActionList := myscoreplayer.score.ActionList
changeSize1 := new ChangeSpriteSizeAction time:9 targetNum:8 \
   width:100 height:120
append myActionList changeSize1
```

Instance Variables

Inherited from `Action`:

```
authorData          targetNum          time
playOnly
```

The following instance variables are defined in `ChangeSpriteAppearanceAction`:

height

self`.height` (read-write) `Number`

The height to be set by the action *self* on the sprite in the channel determined by the value in *self*'s `targetNum` instance variable.

width

self`.width` (read-write) `Number`

The width to be set by the action *self* on the sprite in the channel determined by the value in *self*'s `targetNum` instance variable.

Instance Methods

Inherited from `Action`:

```
trigger
```

ChangeTempoAction

Class type: Tool class (concrete)
Resides in: dirimp.sxl. Works in the KMP and the ScriptX executable.
Inherits from: Action

The ChangeTempoAction class is a specialized subclass of the Action class that represents tempo changes in an animation. When translating a Macromedia Director movie score or animation, the importer creates an instance of the ChangeTempoAction class whenever the tempo of the score changes.

You can add ChangeTempoAction instances to the action list of a Score to add additional sounds to an imported animation.

Creating New ChangeTempoAction Instances

To create a new instance of the ChangeTempoAction class, call the new method, which takes time and tempo keywords.

time:	Time for the sound to start
tempo:	An integer indicating the tempo (that is, the playback rate) for the animation.

Note that this action does not take a targetNum keyword since it works on the score as a whole, not on an individual object in the animation. For example, to set the tempo to 15 at time 3, and then speed it up even more to 20 at time 8:

```
-- myScorePlayer is an existing ScorePlayer
t1:= new ChangeTempoAction time:3 tempo:15
t2:= new ChangeTempoAction time:8 tempo:20
myActionList := myscoreplayer.score.ActionList
append myActionList t1
append myActionList t2
```

Instance Variables

Inherited from Action:

```
authorData          targetNum          time
playOnly
```

The following instance variables are defined in ChangeTempoAction:

tempoValue

self.tempoValue (read-write) Number

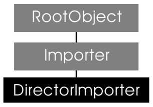
The playback rate to which the ChangeTempoAction *self* changes the associated Score object.

Instance Methods

Inherited from Action:

```
trigger
```

DirectorImporter

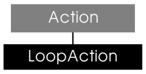


Class type: Tool class (concrete)
Resides in: `dirimp.sxl`. Works in the KMP and the ScriptX executable.
Inherits from: `Importer`

A `DirectorImporter` object imports either a castlist from a Director animation, or a score along with the castmembers, into ScriptX.

The `DirectorImporter` class is used by the Import/Export engine. It does not have methods that are accessible by users.

LoopAction



Class type: Tool class (concrete)
Resides in: `dirimp.sxl`. Works in the KMP and the ScriptX executable.
Inherits from: `Action`

The `LoopAction` class is a specialized subclass of the `Action` class that causes an animation to start playing over again when it reaches the end. Users do not need to create loop actions. To cause an animation to loop, set the `looping` instance variable of the score player to `true`.

Instance Variables

Inherited from `Action`:

`authorData`
`playOnly`
`targetNum`
`time`

The following instance variables are defined in `LoopAction`:

looping

self.`looping` (read-write) `Boolean`

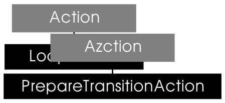
Either `true` or `false` depending on whether the score action player is set to be looping or not. (The score player is the one whose score's action list includes the loop action *self*.)

Instance Methods

Inherited from `Action`:

`trigger`

PrepareTransitionAction



Class type: Tool clas (concrete)
Resides in: `dirimp.sxl`. Works in the KMP and the ScriptX executable.
Inherits from: `Action`

The `PrepareTransitionAction` class is a specialized subclass of the `Action` class that prepares a transition to play. When the transiton starts playing, the score ticker for the animation stops ticking until the transition has finished.

When importing a Director title to a score player, the importing process creates two actions for each transition in the score:

A transition generates two actions:

- A `PrepareTransitionAction`
- A `ScriptAction`

In between these two actions in the action list are the actions that setup the current frame.

The script for the `ScriptAction` is an exported global function called `ScorePlayerStartTransition` that starts the transition playing.

Instance Variables

Inherited from `Action`:

`authorData`	`targetNum`	`time`
`playOnly`		

The following instance variables are defined in `PrepareTransitionAction`:

direction

self`.direction` (read-write) `NameClass`

Specifies the direction for the transition player that will be created by the action *self*.

duration

self`.duration` (read-write) `Number`

Specifies the duration for the transition player that will be created by the action *self*.

scale

self`.scale` (read-write) `Number`

Specifies the scale for the transition player that will be created by the action *self*.

scoreTicker

self`.scoreTicker` (read-write) `ScoreTicker`

Specifies the score ticker that needs to be restarted when the transition started by the action *self* is completed.

When a transtion starts, the animation is suspended, that is, its score ticker (clock) is stopped until the transiton is completed.

transitionClass

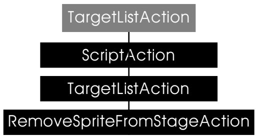
self.`transitionClass` (read-write) `Number`

Specifies the class of transition that the action *self* is preparing to play.

Instance Methods

Inherited from `Action`:
    `trigger`

RemoveSpriteFromStageAction class



Class type: Tool class (concrete)
Resides in: `dirimp.sxl`. Works in the KMP and the ScriptX executable.
Inherits from: `Action`

The `RemoveSpriteFromStageAction` class is a specialized subclass of the `Action` class that makes a sprite disappear from a channel. When translating a Macromedia Director score, the importer creates an instance of the `RemoveSpriteFromStageAction` class whenever a castmember disappears from an animation channel.

Creating New RemoveSpriteFromStageAction Instances

To create a new instance of the `RemoveSpriteFromStageAction` class, call the `new` method, which takes `targetNum` and `time` keywords.

time The time at which the action is to occur.
targetNum The channel which the action affects.

For example, to make make the character in channel 2 disappear at time 5:

```
-- myscoreplayer is an existing ScorePlayer instance
myActionList := myscoreplayer.score.ActionList
spriteDisappear1 := new RemoveSpriteFromStageAction targetNum:2 \
   time:5
prepend myActionList spriteDisappear1
```

Instance Variables

Inherited from `Action`:
`authorData`
`playOnly`
`targetNum`
`time`

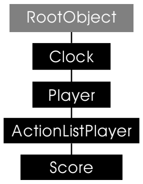
Inherited from `ScriptAction`:
`script`

Inherited from `TargetListAction`:
`rewindScript`

Instance Methods

Inherited from `Action`:
`trigger`

Score



Class type: Tool class (concrete)
Resides in: `dirimp.sxl`. Works in the KMP and the ScriptX executable.
Inherits from: `ActionListPlayer`

The `Score` class is a specialized subclass of the `ActionListPlayer` class that holds the list of actions that make up an animation that was imported from a Director score.

When the Director Importer creates a `ScorePlayer` object, the `ScorePlayer` object creates an instance of `Score` automatically. Note that this instance of `Score` is intended for use by the system to manage the animation, and is not meant to be modified by users. It is used to execute actions at certain times that are set by the score ticker.

The actions that make up the animation are stored in the score's `actionList` instance variable.

Instance Variables

Inherited from `Clock`:

- `callbacks`
- `effectiveRate`
- `masterClock`
- `offset`
- `rate`
- `resolution`
- `scale`
- `slaveClocks`
- `ticks`
- `time`
- `title`

Inherited from `Player`:

- `audioMuted`
- `dataRate`
- `duration`
- `globalBrightness`
- `globalContrast`
- `globalHue`
- `globalPanOffset`
- `globalSaturation`
- `globalVolumeOffset`
- `markerList`
- `status`
- `videoBlanked`

Inherited from `ActionListPlayer`:

- `actionList`
- `targets`
- `authorData`
- `rewindScripts`

The following instance variables are defined in `Score`:

audioPlayers

self`.audioPlayers` (read-write) `Array`

Specifies an array of two `DigitalAudioPlayer` objects. These digital audio players represent the two sound channels of an animation that was imported from a Director score.

`ChangeSoundAction` instances in the list of actions for the score *self* will change the audio stream associated with one of these audio players.

executingTimeJump

self`.exectutingTimeJump` (read-write) `Array`

Specifies whether or not the score *self* is jumping in time rather than playing.

scorePlayer

self`.scorePlayer` (read-only) `ScorePlayer`

Specifies the `ScorePlayer` object that created the score *self*.

stage

self.stage (read-write) Space

Specifies the Space object that the score *self* uses to display and remove sprite objects. That is, it is the stage on which the animation appears when the score is playing.

Instance Methods

Inherited from Clock:

```
addPeriodicCallback    clockAdded             pause
addRateCallback        clockRemoved           resume
addScaleCallback       effectiveRateChanged   timeJumped
addTimeCallback        forEachSlave           waitTime
addTimeJumpCallback    isAppropriateClock     waitUntil
```

Inherited from Player:

```
addMarker              goToBegin              playPrepare
eject                  goToEnd                playUnprepare
fastForward            goToMarkerFinish       playUntil
getMarker              goToMarkerStart        rewind
getNextMarker          pause                  stop
getPreviousMarker      play
```

Inherited from ActionListPlayer:

```
getMuteChannel         setMuteChannel
```

The following instance methods are defined in Score:

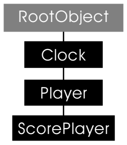
getLast

getLast *self* Action

self.audioPlayers (read-write) Array

Returns the last action in the action list of the score *self*.

ScorePlayer



Class type: Tool class (concrete)
Resides in: `dirimp.sxl`. Works in the KMP and the ScriptX executable.
Inherits from: `Player`

The `ScorePlayer` class is a specialized subclass of the `Player` class that controls the playback of an animation created by importing a Director score. An instance of this class (rather than the `Score` class) is played to run animation.

When the Director Importer imports a Macromedia score, it returns a `ScorePlayer` object that can be used to control the running of an animation.

A `ScorePlayer` object inherits the ability to play, fast forward, rewind, go to a specific marker, pause, and stop playback from the `Player` class. A `ScorePlayer` cannot play backwards.

A score player has a target space (stored in its `stage` instance variable) in which the animation appears when it plays. By default, the stage of a score player is the same size as the stage for the original Director score.

You do need to create instances of the `ScorePlayer` class. When the Director Importer imports a Macromedia Director score, it automatically creates an instance of the `ScorePlayer` class.

If you change the value of the `frame` instance variable of a stopped score player to a frame that has a transition, then the transition will not be set up and will not play when the score player starts playing from that frame.

Instance Variables

Inherited from `Clock`:

`callbacks`
`effectiveRate`
`masterClock`
`offset`
`rate`
`resolution`
`scale`
`slaveClocks`
`ticks`
`time`
`title`

Inherited from `Player`:

`audioMuted`
`dataRate`
`duration`
`globalBrightness`
`globalContrast`
`globalHue`
`globalPanOffset`
`globalSaturation`
`globalVolumeOffset`
`markerList`
`status`
`videoBlanked`

The following instance variables are defined in `ScorePlayer`:

brushCache

self`.brushCache` (read) `Array`

Specifies an array of the brushes used by the text sprites in the animation controlled by the score player *self*.

castList

self`.castList` (read) `Array`

Specifies an array of the 2D sprites, text sprites and audio streams that appear in the animation controlled by the score player self. Each sprite represents a castmember in the Director castlist that was imported.

frame

self.frame (read-write) Integer

Specifies the frame that the score player *self* is currently playing.

loopAction

self.loopAction (read-write) LoopAcation

Specifies the LoopAction instance that is the first action in the action list for the score for the scoreplayer *self*. This loop action is involved in determining whether the score player loops or not. Users never need to interact with the loop action instance or instance variable. Instead, set the value of the score player's looping instance variable to switch looping on and off.

looping

self.looping (read-write) Boolean

Specifies whether or not the animation begins over from the beginning when it reaches the end. If the value is true, the animation loops. If the value is false, the animation stops when it reaches the end.

score

self.score (read-write) Score

Specifies the score containing the list of actions that make up the animation controlled by the score player *self*.

scoreTicker

self.scoreTicker (read) ScoreTicker

Specifies the ScoreTicker object that is a player that has the ScorePlayer as its master clock, and has a callback that calls step or stepBack methods on the ScorePlayer's Score. The ScoreTicker is basically responsible for controlling the timing of the animation.

scoreTickerRate

self.scoreTickerRate (read) Integer

Specifies the rate for the score ticker associated with the score player *self*.

spriteChannelInforray

self.spriteChannelInfoArray (read-write) Array

Specifies an array of 24 or 48 SpriteChannelInfo objects. Each SpriteChannelInfo object is updated during the animation to contain information about the channel it represents, such as the current inkmode for the channel.

stage

self.stage (read-write) Space

A Space or a TwoDMultiPresenter object that the Director Importer created when it initialized the score player *self*. This stage is used to display sprites when the animation plays.

textSpriteCache

self.`textSpriteCache` (read) `Array`

Specifies an array of 24 or 48 `TextSprite` objects. This cache is called upon during animation when text is displayed so that the presenters do not need to be created at animation time.

transitionPlayer

self.`transitionPlayer` (read-write) `Space`

If a transition is playing, specifies the `transitionPlayer` object playing the transition. If no transition is playing, this instance variable has the value `undefined`.

twoDSpriteCache

self.`twoDSpriteCache` (read) `Array`

Specifies an array of 24 or 48 `TwoDSprite` objects. This cache is called upon during animation when images are displayed so that the presenters do not need to be created at animation time.

Instance Methods

Inherited from `Clock`:

`addPeriodicCallback`
`addRateCallback`
`addScaleCallback`
`addTimeCallback`
`addTimeJumpCallback`
`clockAdded`
`clockRemoved`
`effectiveRateChanged`
`forEachSlave`
`isAppropriateClock`
`pause`
`resume`
`timeJumped`
`waitTime`
`waitUntil`

Inherited from `Player`:

`addMarker`
`eject`
`fastForward`
`getMarker`
`getNextMarker`
`getPreviousMarker`
`goToBegin`
`goToEnd`
`goToMarkerFinish`
`goToMarkerStart`
`pause`
`play`
`playPrepare`
`playUnprepare`
`playUntil`
`rewind`
`stop`

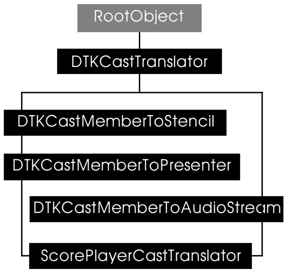
The following instance methods are defined in `ScorePlayer`:

restartAnimation

`restartAnimation` *self* ⇨ `undefined`

Restarts the animation played by the score player *self* when a transtion finishes playing.

ScorePlayerCastTranslator



Class type: Tool class (concrete)
Resides in: `dirimp.sxl`. Works in the KMP and the ScriptX executable.
Inherits from: `DTKCastMemberToPresenter` and `DTKCastMemberToAudioStream`

The `ScorePlayerCastTranslator` class has methods for translating castmembers in a Director score to ScriptX objects. It translates shape and bitmap castmembers to `TwoDSprite` objects, it translates text castmemberts to `TextSprite` objects, and it translates sound castmembers to `AudioStream` objects. It cannot translate video and button cast members, they become `undefined` objects in ScriptX.

The Director importer creates and uses a `ScorePlayerCastTranslator` object. Users should not need to either create or modify instances of this class.

Instance Variables

brushCache

self`.brushCache` (read) `Array`

Specifies an array of the brushes used by `TextSprite` objects that represent converted text castmembers in the castlist converted by *self*.

Instance Methods

translateBitmap

`translateBitmap` *self bitmapCast* ⇨ `TwoDSprite`

self	`ScorePlayerCastTranslator` object
bitmapCast	A `DTKBitmap` object representing a Director bitmap castmember.

Translates a Director bitmap castmember in the castlist being converted by *self* to a `TwoDSprite` object. The 2D sprite has an appropriate bitmap in its `target` instance variable.

translateShape

`translateShape` *self shapeCast* ⇨ `TwoDSprite`

self	`ScorePlayerCastTranslator` object
shapeCast	A `DTKShape` object representing a Director shape castmember.

Translates a Director shape castmember in the castlist being converted by *self* to a `TwoDSprite` object. The 2D sprite has an appropriate stencil in its `target` instance variable.

translateSound

`translateSound` *self soundCast* ⇨ `AudioStream`

self	`ScorePlayerCastTranslator` object
soundCast	A castmember displaying text.

Translates a Director sound castmember in the castlist being converted by *self* to an `AudioStream` object. The audio sprite has an appropriate stream containing digitiazed sound data in its `inputStream` instance variable.

translateText

`translateText` *self textCast* ⇨ `TwoDSprite`

self	`ScorePlayerCastTranslator` object
textCast	A castmember displaying text.

Translates a Director text castmember in the castlist being converted by *self* to a `TextSprite` object. The text sprite has appropriate text in its `target` instance variable.

translateUnknwn

`translateText` *self cast* ⇨ `undefined`

self	`ScorePlayerCastTranslator` object
cast	A castmember of unknown type.

Translates a Director text castmember of unknown type in the castlist being converted by *self* to an undefined object.

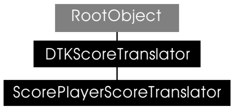
translateVideo

`translateText` *self videoCast* ⇨ `VideoSprite`

self	`ScorePlayerCastTranslator` object
videoCast	A castmember that plays video.

Translates a Director video castmember in the castlist being converted by *self* to a `VideoSprite` object. The video sprite can play the appropriate movie.

ScorePlayerScoreTranslator



Class type: Tool class (concrete)
Resides in: `dirimp.sxl`. Works in the KMP and the ScriptX executable.
Inherits from: `DTKScoreTranslator`

A `ScorePlayerScoreTranslator` object translates a Director score into a ScriptX `ScorePlayer` object and other associated objects.

You do not need to create an instance of the `ScorePlayerScoreTranslator` class. When the Director Importer imports a Director score, it creates an instance of the `ScorePlayerScoreTranslator` class automatically.

Instance Variables

Inherited from `DTKScoreTranslator`:

`castlist` `container` `dtk`
`outputStream`

The following instance variables are defined in `ScorePlayerScoreTranslator`:

scorePlayer

self`.scorePlayer` (read-write) `ScorePlayer`

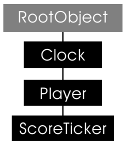
Specifies the `ScorePlayer` object that represents the Director score translated into ScriptX by the translator *self*.

Instance Methods

Inherited from `DTKScoreTranslator`:

`prepareToTranslateScore`
`translateFrame`
`setDTK`

ScoreTicker



Class type: Tool class (concrete)
Resides in: `dirimp.sxl`. Works in the KMP and the ScriptX executable.
Inherits from: `Player`

A `ScoreTicker` instance is a clock that ticks a `Score`.

Each imported Director animation has a `ScoreTicker` object associated with the `Score` object for the animation. The score ticker is intended for use by the system to manage the timing of the animation, and is not meant to be modified by users.

Instance Variables

See the *ScriptX Class Reference* for the instance variables inherited from superclasses of `ScoreTicker`.

The following instance variables are defined in `ScoreTicker`:

stepCallback

self`.stepCallback` (read-write) `Callback`

Specifies the callback that calls the `step` or `stepBack` methods on the score that is ticked by the score ticker *self*.

stepTarget

self`.stepTarget` (read-write) `Score`

Specifies the target to which to send the current method. The target is the score that is ticked by the score ticker *self*.

Instance Methods

See the *ScriptX Class Reference* for the instance methods inherited from superclasses of `ScoreTicker`.

The following instance methods are defined in `ScoreTicker`:

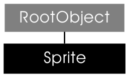
tickleAnimation

`tickleAnimation` *self score* ⇨ `target`

self	`ScoreTicker` object
score	The `Score` object to be ticked.

Invokes the callback in the `stepCallback` instance variable to call the appropriate step method on the score that is ticked by the score ticker *self*.

Sprite

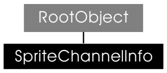


Class type: Tool class (abstract)
Resides in: `dirimp.sxl`. Works in the KMP and the ScriptX executable.
Inherits from: `RootObject`

This is an abstract class, whose purpose is to provide a common superclass to the classes `TwoDSprite,` `TextSprite`, and `VideoSprite`.

When a Director castlist is imported into ScriptX, each castmember becomes a a 2D sprite, a text sprite, or a video sprite.

SpriteChannelInfo



Class type: Tool class (concrete)
Resides in: `dirimp.sxl`. Works in the KMP and the ScriptX executable.
Inherits from: `RootObject`

A `SpriteChannelInfo` instance holds information about the state of a channel during an animation playback. Currently a `SpriteChannelInfo` object records the `channelNumber`, `inkMode`, `matteColor` and `invisibleColor`.

When the Director Importer imports a Director score, it creates 24 or 48 `SpriteChannelInfo` objects – one for each channel. The `ScorePlayer` created by the importing process has an instance variable `SpriteChannelInfo`, whose value is an array of all the `SpriteChannelInfo` objects for this animation.

As the animation progresses, each `SpriteChannelInfo` object updates to maintain the correct information about the channel for the current time.

Instance Variables

These instance variables are for use by the system to maintain information about the state of the animation, and should not be changed by users.

channelNumber

self`.channelNumber` (read) `Integer`

The channel that the `SpriteChannelInfo` object *self* holds information about.

inkMode

self`.inkMode` (read-write) `NameClass`

The current inkmode for this channel, which will be either `@copy`, `@matte`. or `@invisible`.

matteColor

self`.matteColor` (read-write) `RGBColor`

The current matte color for this channel.

invisibleColor

self`.invisibleColor` (read-write) `RGBColor`

The current invisible color for this channel.

Instance Methods

setInk

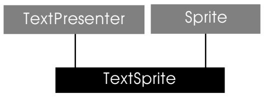
`setInk` *self* *newInk* ⇨ `NameClass`

self	`SpriteChannelInfo` object
newInk	The new inkmode.

Updates the `SpriteChannelInfo` object *self* to hold the current inkmode information.

Users should not call this method. It gets called automatically at the appropriate time.

TextSprite



Class type: Tool class (concrete)
Resides in: `dirimp.sxl`. Works in the KMP and the ScriptX executable.
Inherits from: `Sprite` and `TextPresenter`

The class `TextSprite` represents sprites in an animation that display text.

When the Director importer imports a Director castlist into ScriptX, each castmember that displays text becomes a `TextSprite` instance. To speed up the running of an animation, the `ScorePlayer` keeps a cache of 24 or 48 `TextSprite` instances, one for each channel, which are used to present text on the screen during the animation.

The `TextSprite` class inherits instance variables from its superclasses. For example, the `boundary` instance variable denotes the shape to be presented; the `width` instance variable denotes the width; the `height` instance variable denotes the height; and the `x` and `y` instance variable denotes the x and y values of the shape's position. The `z` instance variable denotes the z ordering, which determines where the shape is in the stack of displayed images (an image with a high z value appears on top of an image with a low z value on the screen.)

To change the values of instance variables of `TextSprite` instances that are participating in an animation, you should use appropriate `Action` instances. For example, use a `PathAction` to change the position; use a `ChangeInkAction` to change the ink mode and use a `ChangeSpriteSizeAction` to change the size.

Instance Variables

Inherited from `Presenter`:

`presentedBy`
`subPresenters`
`target`

Inherited from `TwoDPresenter`:

`bBox`
`boundary`
`changed`
`clock`
`compositor`
`direct`
`eventInterests`
`globalBoundary`
`globalRegion`
`globalTransform`
`height`
`imageChanged`
`isVisible`
`position`
`stationary`
`target`
`transform`
`width`
`window`
`x`
`y`
`z`

Inherited from `TextPresenter`:

`attributes`
`cursor`
`cursorBrush`
`enabled`
`fill`
`inset`
`offset`
`selectionBackground`
`selectionForeground`
`stroke`

The following instance variables are defined in `TextSprite`:

invisibleColor

self`.invisibleColor` (read) `RGBColor`

The invisible color for the text sprite *self*.

matte

self`.matte` (read) `RGBColor`

The matte color for the text sprite *self*.

Instance Methods

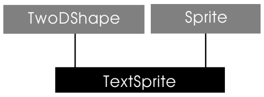
Inherited from `TwoDPresenter`:

`addChangedRegion`
`adjustClockMaster`
`createInterestList`
`hide`
`draw`
`getBoundaryInParent`
`localToWindow`
`compositorChanged`
`refresh`
`windowToLocal`
`show`

Inherited from `TextPresenter`:

`calculate`
`getOffsetForXY`
`getPointForOffset`
`processMouseDown`

TwoDSprite



Class type: Tool class (concrete)
Resides in: dirimp.sxl. Works in the KMP and the ScriptX executable.
Inherits from: Sprite and TwoDShape

The class TwoDSprite represents sprites in an animation that display a shape or a bitmap

When the Director importer imports a Director score into ScriptX, each shape castmember or bitmap castmember becomes a TwoDSprite instance.To speed up the running of an animation, the ScorePlayer keeps a cache of 24 or 48 TwoDSprite instances, one for each channel, which are used to present images on the screen during the animation.

The TwoDSprite class inherits instance variables from its superclasses. For example, the boundary instance variable denotes the shape to be presented; the width instance variable denotes the width; the height instance variable denotes the height; and the x and y instance variable denotes the x and y values of the shape's position. The z instance variable denotes the z ordering, which determines where the shape is in the stack of displayed images (an image with a high z value appears on top of an image with a low z value on the screen.)

To change the values of instance variables of TwoDSprite instances that are participating in an animation, you should use appropriate Action instances. For example, use a PathAction to change the position; use a ChangeInkAction to change the ink mode and use a ChangeSpriteSizeAction to change the size.

Instance Variables

Inherited from Presenter:

presentedBy
subPresenters
target

Inherited from TwoDPresenter:

bBox
boundary
changed
clock
compositor
direct
eventInterests
globalBoundary
globalRegion
globalTransform
height
imageChanged
isVisible
position
stationary
target
transform
width
window
x
y
z

Inherited from TwoDShape:

fill
stroke

The following instance variables are defined in TwoDSprite:

invisibleColor

self.invisibleColor (read) RGBColor

The invisible color for the 2D sprite *self*.

matteColor

self.matte (read) RGBColor

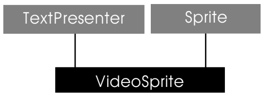
The matte color of the 2D sprite *self*.

Instance Methods

Inherited from `TwoDPresenter`:

`addChangedRegion`
`adjustClockMaster`
`createInterestList`
`hide`
`draw`
`getBoundaryInParent`
`localToWindow`
`compositorChanged`
`refresh`
`windowToLocal`
`show`

VideoSprite



Class type: Tool class (concrete)
Resides in: dirimp.sxl. Works in the KMP and the ScriptX executable.
Inherits from: Sprite and InterleavedMoviePlayer

The class VideoSprite represents sprites in an animation that play video.

When the Director importer imports a Director castlist into ScriptX, each castmember that plays video becomes a VideoSprite instance.

The TextSprite class inherits instance variables from its superclasses. For example, the boundary instance variable denotes the shape to be presented; the width instance variable denotes the width; the height instance variable denotes the height; and the x and y instance variable denotes the x and y values of the shape's position. The z instance variable denotes the z ordering, which determines where the shape is in the stack of displayed images (an image with a high z value appears on top of an image with a low z value on the screen.)

To change the values of instance variables of TextSprite instances that are participating in an animation, you should use appropriate Action instances. For example, use a PathAction to change the position; use a ChangeInkAction to change the ink mode and use a ChangeSpriteSizeAction to change the size.

Instance Variables

See the *ScriptX Class Reference* for the instance variables inherited from superclasses of VideoSprite.

The following instance variables are defined in VideoSprite:

invisibleColor

self.invisibleColor (read) RGBColor

The invisible color for the video sprite *self*.

matte

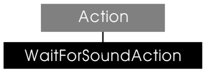
self.matte (read) RGBColor

The matte color for the video sprite *self*.

Instance Methods

See the *ScriptX Class Reference* for the instance methods inherited from superclasses of VideoSprite.

WaitForSoundAction



Class type: Tool class (concrete)
Resides in: `dirimp.sxl`. Works in the KMP and the ScriptX executable.
Inherits from: `Action`

The `WaitForSoundAction` class is a specialized subclass of the `Action` class that makes an animation wait until a sound has finished before it resumes.

When translating a Macromedia Director score, the importer creates an instance of the `WaitForSoundAction` class whenever the score has a WaitForSound tempo control.

Creating New WaitForSoundAction Instances

To create a new instance of the `WaitForSoundAction` class, call the `new` method, which takes `time` and `channelNum` keywords.

time: Time to start waiting.
channelNumber: An integer, either 1 or 2, indicating which sound channel is playing the sound that the animation should wait for to be finished.

Note that this action does not take a `targetNum` keyword since it works on the score as a whole, not on any individual object in the animation. For example, to tell the animation to stop playing at time 10 and to wait until the sound in sound channel 1 has finished before resuming:

```
-- myScorePlayer is an existing ScorePlayer
wait1:= new WaitForSoundAction time:10 channelNumber:1
myActionList := myscoreplayer.score.ActionList
append myActionList wait1
```

Instance Variables

Inherited from `Action`:

```
authorData          targetNum          time
playOnly
```

The following instance variables are defined in `WaitForSoundAction`:

channelNumber

self.`channelNumber` (read) `RGBColor`

The number of the sound channel which this action is waiting for. The value will be either 1 or 2.

channelNumber

self.`channelNumber` (read) `RGBColor`

The number of the sound channel which this action waits for. The value is either 1 or 2, which is an index into the array in the `audioPlayers` instance variable of the score containing the action *self*.

scoreTicker

self.`scoreTicker` (read-write) `scoreTicker`

Specifies the score ticker that needs to be restarted when the sound for wich the action *self* is waiting is completed.

When the `waitForSoundAction` is invoked, the animation is suspended, that is, its score ticker (clock) is stopped until the sound is completed.

scoreTickerRate

self`.scoreTickerRate` (read-write) `scoreTicker`

Specifies the rate at which to restart the score ticker discussed above.

Instance Methods

Inherited from `Action`:
    `trigger`

The following instance methods are defined in `WaitForSoundAction`:

startScoreTicker

`startScoreTicker` *self* ⇨ `undefined`

Starts the score ticker ticking again when the sound for which the animation is waiting has finished playing. The result is that the animation starts playing again.

CHAPTER

18

Using The Director Translation Kit

This chapter discusses the use of the Director Translator Kit (DTK) to build customized importers for Director titles.

The Director to ScorePlayer Importer, discussed in Chapter 16, "Using the Director-to-ScorePlayer Importer", is one example of an importer that can be built with the Director Translation Kit.

If you want to import a simple Director animation, and play it back in ScriptX automatically, then the Director to ScorePlayer Importer may be sufficient for your needs, and you do not need to read this chapter. However, if you have your own ideas about how to recreate a Director title in ScriptX and you do not want to use the `ScorePlayer` mechanism provided by the Director to ScorePlayer Importer, then you need to write your own customized Director Translator. If that's the case, you should read this chapter and you should also be very familiar with how to use the ScriptX language.

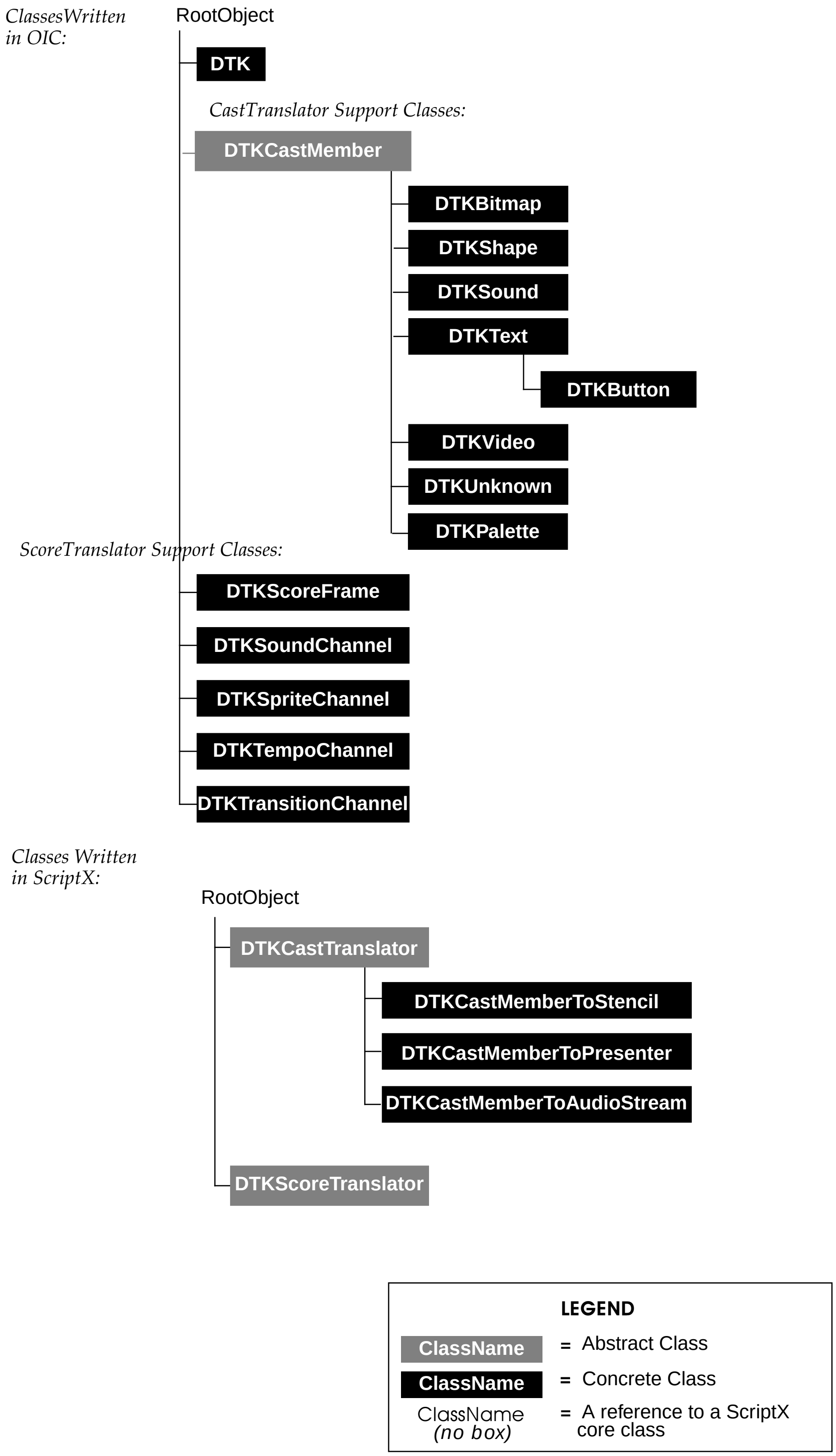
The translation of data from a Director title to ScriptX-readable data is managed by a `DTK` object. The `DTK` object is simply a translator — it translates a description of the Director animation into ScriptX-readable information. It does not interpret the information or work out how to reproduce the animation in ScriptX.

During the process of translating a Director title, the `DTK` object gets the information from the Director title and makes it available to customized ScriptX cast and score translators. These translators in turn convert the data into ScriptX objects that recreate the Director title in ScriptX. You must write the translators.

This chapter assumes you have intimate knowledge of Director castlists and scores. It does not explain concepts or terms that would be familiar to experienced Director users. Chances are that if you're not experienced at Director then you're not ready to write your own customized Director importers.

Classes and Inheritance

This section shows the inheritance hierarchy for the classes in the Director Translator Kit. It also lists and provides brief descriptions of these classes.


ClassesWritten in OIC:
RootObject
DTK
CastTranslator Support Classes:
DTKCastMember
DTKBitmap
DTKShape
DTKSound
DTKText
DTKButton
DTKVideo
DTKUnknown
DTKPalette
ScoreTranslator Support Classes:
DTKScoreFrame
DTKSoundChannel
DTKSpriteChannel
DTKTempoChannel
DTKTransitionChannel
Classes Written in ScriptX:
RootObject
DTKCastTranslator
DTKCastMemberToStencil
DTKCastMemberToPresenter
DTKCastMemberToAudioStream
DTKScoreTranslator
LEGEND
ClassName = Abstract Class
ClassName = Concrete Class
ClassName (no box) = A reference to a ScriptX core class

DTK Class and the Classes that Hold Raw Data About a Director Animation

These loadable classes are written in Objects In C (OIC), the language used to write the ScriptX core classes. The DTK class, the DTKCastMember class and the subclasses of DTKCastMember are fully defined. You do not need to subclass or modify them. These classes retrieve the data from a Director animation and store it in a way that other ScriptX classes (namely DTKCastTranslator and DTKScoreTranslator classes) can access and use.

In the following list of classes, indentation indicates inheritance.

DTK— has methods that control the process of importing a Macromedia Director animation into ScriptX.

DTKCastMember—an abstract class that is the common superclass for all classes that hold information about Director castmembers.

> DTKBitmap—holds information about bitmap castmembers.
>
> DTKShape—holds information about shape castmembers.
>
> DTKSound—holds information about sound castmembers.
>
> DTKText—holds information about text castmembers.
>
>> DTKButton—holds information about button castmembers.
>
> DTKUnknown—holds information about castmembers of an unknown kind.
>
> DTKVideo—holds information about video castmembers.
>
> DTKPalette—holds information about colormap castmembers.

DTKScoreFrame—holds information about a frame of a Macromedia Director score.

DTKSoundChannel—holds information about a sound channel for a frame of a Macromedia Director score.

DTKSpriteChannel—holds information about a sprite channel for a frame of a Macromedia Director score.

DTKTempoChannel—holds information about a tempo channel for a frame of a Macromedia Director score.

DTKTransitionChannel—holds information about a transition channel for a frame of a Macromedia Director score.

Cast Translator and Score Translator Classes

These loadable classes are written in ScriptX. These are the classes you need to subclass and modify to create your own customized Director importer.

In the following list of classes, indentation indicates inheritance.

DTKCastTranslator—the common superclass for all classes that convert Director cast members to ScriptX objects.

> DTKCastMemberToAudioStream—takes information from a DTKSound object and converts it to an AudioStream object.
>
> DTKCastMemberToStencil—takes information from a DTKShape, DTKBitmap or DTKText object and converts it to a Stencil or Text object in ScriptX.

DTKCastMemberToPresenter—takes information from a DTKShape, DTKBitmap, a DTKText or DTKVideo object and converts it to a TwoDShape, TextPresenter or InterleavedMoviePlayer object as appropriate.

DTKScoreTranslator—converts a Director score to ScriptX objects.

How the Director Translation Process Works

The process of translating a Director title into a ScriptX title is managed by a DTK instance, which has two helpers, a DTKCastTranslator instance and a DTKScoreTranslator. A DTK instance's castTranslator instance variable holds the cast translator, and the scoreTranslator instance variable holds the score translator.

The DTK instance's translateDirector method controls the translation, which has two parts:

- Translating the castmembers.

  The translateDirector method calls the translateCast method. The translateCast method examines each cast member in turn and determines what kind it is (such as bitmap, sound, text or button), then passes information about the castmember to the DTKCastTranslator instance. The cast translator determines how the castmember is implemented as ScriptX objects and instance variables. See "Cast List Translation" on page 261 for details.

- Translating the score.

  The translateDirector method then calls the translateScore method. The translateScore method passes information about each frame in the score in turn to the translateFrame method on the score translator. The translateFrame method on the score translator determines how the information is implemented as ScriptX objects and instance variables. See "Score Translation" on page 262 for details.

The DTK object translates the cast if it has a cast translator in its castTranslator instance variable, and it translates the score if it has a score translator in its scoreTranslator instance variable. Thus you can choose to translate just the cast list, just the score, or both the cast list and the score, by setting either or both of the castTranslator and scoreTranslator instance variables.

Cast List Translation

A DTK instance's translateDirector method calls the translateCast method to translate a Director cast list into ScriptX.

During cast translation, the DTK instance works with the DTKCastTranslator in its castTranslator instance variable to translate each castmember.

The translateCast method first calls the prepareToTranslateCast method on the cast translator. Then it does does the following for each castmember:

1. Figures out what media type the castmember is and creates an instance of the appropriate subclass of the DTKCastMember class.

   If the castmember is a bitmap, the DTK creates a DTKBitmap instance.

   If the castmember is a shape (such as an oval, rectangle or round rectangle) the DTK creates a DTKShape instance.

   If the castmember is a sound, the DTK creates a DTKSound instance.

   If the castmember is a text box, the DTK creates a DTKText instance.

   If the castmember is a button, the DTK creates a DTKButton instance.

   If the castmember is a movie, the DTK creates a DTKVideo instance.

   If the castmember is a palette, the DTK creates a DTKPalette instance.

   If the castmember is anything else for now, the DTK creates a DTKUnknown instance.

2. Calls a method on the cast translator, passing in the new object as an argument.

   If the castmember is a bitmap, the DTK calls the translateBitmap method on the cast translator, passing it the newly created DTKBitmap instance.

   If the castmember is a shape (such as an oval, rectangle or round rectangle) the DTK calls the translateShape method on the cast translator, passing it the newly created DTKShape instance.

   If the castmember is a sound, the DTK calls the translateSound method on the cast translator, passing it the newly created DTKSound instance.

   If the castmember is a text box, the DTK calls the translateText method on the cast translator, passing it the newly created DTKText instance.

   If the castmember is a button, the DTK calls the translateButton method on the cast translator, passing it the newly created DTKButton instance.

   If the castmember is a movie, the DTK calls the translateVideo method on the cast translator, passing it the newly created DTKVideo instance.

   If the castmember is a palette, the DTK calls the translatePalette method on the cast translator, passing it the newly created DTKPalette instance.

   If the castmember is anything else for now, the DTK calls the translateUnknown method on the cast translator, passing it the newly created DTKUnknown instance.

Score Translation

A DTK instance's translateDirector method calls the DTK's translateCast method, then calls the translateScore method to translate the Director score into ScriptX.

The translateScore method works with the DTKScoreTranslator instance in the scoreTranslator instance variable.

The translateScore method does the following:

1. First it copies the value of the castList instance variable on the DTK instance to the castList instance variable on the score translator. This enables the score translator to have its own record of the objects that represent the castmembers. If the cast list has not already been translated, this instance variable will be empty.
2. Next it calls the prepareToTranslateScore on the score translator.
3. Then it starts translating the data for each frame on a frame by frame basis.

   At each frame, it calls the translateFrame method on the score translator, passing information about the frame in the form of two DTKScoreFrame objects, one for the current frame and one for the previous frame. It also passes a *changedArray* argument, indicating the sprite channels in which changes have occurred this frame.

   The translateFrame method determines what to do with the information and how to turn it into ScriptX objects and instance variables. You must write the translateFrame method.

What You Must Do To Build a Customized Director Importer

This section lists the steps for building your own, completely new Director Translator. Each step is discussed in detail in a section of its own following this listing.

1. Design a Paradigm for Recreating a Director Title in ScriptX

Work out how you want to recreate the castmembers and actions of the score in ScriptX, and, if necessary, create the classes to support the paradigm. What these classes are and do is entirely up to you.

2. Load the Director Translation Kit

The files needed for the Director Translation Kit are contained in a folder called DTK. Make sure that the DTK folder is in the same folder as your ScriptX executable.

To load the Director Translation Kit, open the title container saved to the file dtk.sxl in the dirimp directory.

```
open TitleContainer path:"dtk/dtk.sxl"
```

You can also use the **Open** command from the File menu to open the title container.

When this file has finished loading, it leaves you in the Scratch module. However, to use the Director Translation Kit, you must work in the DTK module.

The best way to handle the module-switching is to create modules of your own that use the necessary modules.

3. Define Your Customized Cast Translator Class

Create a subclass of DTKCastTranslator and define its appropriate media translation methods (that is translateBitmap, translateShape, translateText, translateSound, translateVideo and translateButton).

Define the prepareToTranslate Cast method on your cast translator class to do any work (such as printing messages) that needs to be done before cast translation begins.

Create the new class as a subclass of DTKCastTranslator if you want to define all the media translation methods from scratch, or create it as a subclass of one or more existing subclasses of DTKCastTranslator if you want it to inherit some predefined functionality. See "Choosing Superclasses for Your New Cast Translator Class" on page 265 for more details of the functionality provided by the existing subclasses of DTKCastTranslator.

The input argument to a media translation method is a DTKCastMember object that contains information about the castmember to be translated:

- The argument to translateBitmap is a DTKBitmap object.
- The argument to translateShape is a DTKShape object.
- The argument to translateText is a DTKText object.
- The argument to translateSound is a DTKSound object.
- The argument to translateButton is a DTKButton object.
- The argument to translatePalette is a DTKPalette object.
- The argument to translateVideo is a DTKVideo object.
- The argument to anything else is a DTKUnknown object.

The media translation methods can access instance variables on the DTKCastMember instance. See "Accessing Information about the Director Castmembers" on page 268 for a list of instance variables for each subclass of DTKCastMember.

The returned value of each media translation method is automatically added to the castlist instance variable of the DTK instance associated with the cast translator, so make sure that each media translation method returns a value that reasonably represents the castmember.

4. Define Your Customized Score Translator Class

Create a subclass of DTKScoreTranslator and define its prepareToTranslateScore and translateFrame methods.

The prepareToTranslateScore method takes no arguments other than *self*. Define this method to do any prelimanary work necessary before translating the score.

The translateFrame method takes four arguments: *self*, the previous frame, the current frame, and a "changed array."

The previous frame and current frame are both DTKScoreFrame objects that contain information about the relevant frames.

The changed array argument is a 24 member array for Director 3.1 titles or a 48 member array for Director 4.0 titles. Each value in the array is a boolean representing whether or not a change has occured in each sprite channel. For example, if sprite channel 5 is different in any way in this frame than in the previous frame, then changedArray[5] is true. If there is no change, changedArray[5] is false.

The translateFrame method must do all the work of taking the raw frame data and turning it into useful ScriptX objects. You must write the translateFrame method.

The translateFrame method can retrieve information about the current frame by accessing instance variables on the DTKScoreFrame object for the current frame. See “Accessing Information about a Frame of the Director Score” on page 269 for a list of instance variables on a DTKScoreFrame object.

The translateFrame method can figure out what’s different between the two frames by comparing the FrameScore objects and then do appropriate tasks in keeping with your paradigm for recreating Director titles in ScriptX.

5. Import a Director Animation into ScriptX

After you have created your own customized class of cast translators and score translators, you are ready to translate a Director title, as follows:

1. Create three instances:
   - an instance of the class DTK
   - an instance of your customized cast translator class
   - an instance of your customized score translator class
2. Set the cast translator instance as the value of the DTK instance’s castTranslator instance variable, and set the score translator as the value of the scoreTranslator instance variable.

   To translate the cast only, set only the cast translator. To translate the score only, set only the score translator.
3. Start the translation by calling the translateDirector method on the DTK instance.

See “Importing a Director Animation into ScriptX” on page 274 for example code for creating DTK, cast and score translator instances and setting the translation in motion.

Designing a Paradigm for Recreating a Director Title in ScriptX

When building a customized Director Translator the first step is to design the paradigm for how the Director score and castlist should be implemented in ScriptX, and build the support classes to implement the paradigm.

For example, the Director to ScorePlayer Importer uses a ScorePlayer class, Score class, customized Action classes, and Sprite classes to represent a Director animation.

Although the subject of designing your own paradigm takes up just a few paragraphs in this document, it is no small task. The reason there is so little documentation for it is that you have to design it and do all the work for it yourself and there is no set way to go about doing that.

Defining Customized Cast Translator Classes

To specify how to translate Director cast members into ScriptX objects, create a customized subclass of `DTKCastTranslator` and define the methods to translate the media used by the Director cast member.

`DTKCastTranslator` is the pre-defined superclass for all cast translators. It contains the `translateBitmap`, `translateShape`, `translateText`, `translateSound`, `translateButton`, `translateVideo`, `translatePalette` and `translateUnknown` methods. For the duration of this chapter, these methods are referred to collectively as the media translation methods.

The value returned by a media translation method is automatically installed as the next member of the array in the `castList` instance variable of the `DTK`. This castlist is often used by the score translator when the time comes to translate the score. You do not need to write the code to install the returned value into the castlist, but you do need to make sure that the media translation methods return values that are meaningful to add to the `DTK`'s castlist.

The class `DTKCastTranslator` has an instance variable called `outputStream`. The intent of this instance variable is to point to an output stream associated with the cast translator. The media translation methods on the cast translator can, if desired, send information about the translated castmember to the output stream. (If not defined, the output stream defaults to the debug stream.)

Choosing Superclasses for Your New Cast Translator Class

The `DTKCastTranslator` class defines all the media translation methods to simply return `undefined`. However, three subclasses of `DTKCastTranslator` are provided to give some level of pre-defined behavior for cast translators.These are:

- `DTKCastMemberToStencil` class

  This class defines the `translateShape` method to convert Director bitmap castmembers to ScriptX `Bitmap` objects and Director shape castmembers to ScriptX `Stencil` objects. It defines the `translateText` method to convert Director text castmembers to ScriptX `Text` objects.

- `DTKCastMemberToPresenter` class

  This class defines the `translateShape` method to convert Director bitmap and shape castmembers to `TwoDShape` objects; to convert text castmembers to `TextPresenter` objects; and to convert video castmembers to InterleavedMoviePlayer objects.

- `DTKCastMemberToAudioStream` class

  This class defines the `translateSound` method to convert Director sound castmembers to ScriptX `AudioStream` objects.

If you want to define all the media translation methods from scratch then create your class as a subclass of `DTKCastTranslator`. The more likely scenario is that you will want to use some of the predefined cast translation behavior, and add some behavior of your own design. In this case, pick the existing subclasses of cast translator whose functionality is closest to the desired functionality, to be the superclasses of your customized cast translator class. Modify the media translation methods to suit your needs.

A media translation method can call `nextMethod` to invoke the superclass's method. For example, suppose you want your cast translator to translate Director sound castmembers to digital audio players and to translate Director bitmap, shape, text and video castmembers to appropriate `Presenter` objects that know the name of the cast member. You also want send details about the cast member to an output file.

In this case you would create subclasses of the appropriate kinds of Presenter and give each subclass a name instance variable. You would also need to create a new cast translator class, whose superclasses are DTKCastMemberToAudioStream and DTKCastMemberToPresenter.(Later on, when you create an instance of the cast translator, you will need to set its textClass, shapeClass, bitmapClass, and videoClass instance variables to the appropriate class.)

You would define the translateSound method on the cast translator class to return an instance of your new digital audio player class instead of an audio stream. You would define the translateShape, translateBitmap, translateText, translateVideo, and translateSound methods to call nextMethod, and to install the name of the castmember in the name instance variable, as well as sending the information to an outp file. The sample code is shown below. (You can also find the code in the DTKsampl folder on the CD.)

```
in Module myModule

-- create subclass that have a name iv
class CastTwoDShape (TwoDShape)
instance variables
   name
end

class CastTextPresenter (TextPresenter)
instance variables
   name
end

class CastMoviePlayer (InterleavedMoviePlayer)
instance variables
   name
end

class CastDigitalAudioPlayer (DigitalAudioPlayer)
instance variables
   name
end

-- create a new class of CastTranslator
class CastToInfoTranslator (DTKCastMemberToAudioStream,
                            DTKCastMemberToPresenter)
end

-- When you create an instance of tstMemberToInfo translator,
-- you will need to set the following ivs:
-- bitmapClass := castTwoDShape
-- shapeClass := castTwoDShape
-- textClass := CastTextPresenter
-- videoClass := CastMoviePlayer

method translateShape self {class CastToInfoTranslator} shapeCast ->
(
   local thePresenter := nextMethod self shapeCast
   thePresenter.name := shapeCast.castName

   format self.outputStream "\nCast member at position %1 is a shape named %2.
   It is being translated to a CastTwoDShape. \n" \
      #(shapeCast.castNumber, shapeCast.castName) #(@unadorned, @unadorned)

   -- return the presenter so that it is put in the cast list
   thePresenter
)


method translateBitMap self {class CastToInfoTranslator} bitmapCast ->
(
   local thePresenter := nextMethod self bitmapCast
   thePresenter.name := bitmapCast.castName

   format self.outputStream "\nCast member at position %1 is a bitmap named %2.
   It is being translated to a CastTwoDShape. \n" \
      #(bitmapCast.castNumber, bitmapCast.castName) #(@unadorned, @unadorned)

   -- return the presenter so that it is put in the cast list
   thePresenter
)


method translateText self {class CastToInfoTranslator} textCast ->
(
   local thePresenter := nextMethod self textCast
   thePresenter.name := textCast.castName

   format self.outputStream "\nCast member at position %1 is a text item
   named %2.It is being translated to a CastTextPresenter. \n" \
      #(textCast.castNumber, textCast.castName) #(@unadorned, @unadorned)

   -- return the presenter so that it is put in the cast list
   thePresenter
)


method translateVideo self {class CastToInfoTranslator} videoCast ->
(
   local thePresenter := nextMethod self videoCast
   thePresenter.name := videoCast.castName

   format self.outputStream "\nCast member at position %1 is a video named %2.
   It is being translated to a CastMoviePlayer. \n" \
      #(videoCast.castNumber, videoCast.castName) #(@unadorned, @unadorned)
   -- return the presenter so that it is put in the cast list
   thePresenter
)


method translateSound self {class CastToInfoTranslator} soundCast ->
(
   local theAudioStream := nextMethod self soundCast
   local thePlayer := new CastDigitalAudioPlayer mediaStream:theAudioStream

   format self.outputStream "\nCast member at position %1 is a sound named %2.
   It is being translated to a cast digital audio player. \n" \
      #(soundCast.castNumber, soundCast.castName) #(@unadorned, @unadorned)

   -- return the player so that it is put in the cast list
```

```
    thePlayer
)
```

Accessing Information about the Director Castmembers

For each castmember in the castlist, the DTK object’s translateCast method figures out what media type the castmember is and creates an instance of DTKBitmap, DTKShape, DTKText, DTKSound, DTKButton, DTKVideo or DTKUnknown as appropriate. The DTK passes this instance to the appropriate media translation method on its cast translator. The media translation methods can get information about the castmember being processed by accessing instance variables on the object passed to it.

The following list gives the names of the instance variables for the class DTKCastMember and its subclasses DTKBitmap, DTKShape, DTKText, DTKSound, DTKButton, DTKVideo and DTKUnknown.

See the Chapter 19, “Director Translation Kit API” reference for more information on each of these instance variables.

- DTKCastMember class

  ```
  castFileName     castFileType     castFullPath   castName
  castNumber       container        lingoScript    linked
  ```

- DTKBitmap class

  ```
  bbox             registrationpt   bitmap
  ```

- DTKText class

  ```
  backGroundColor  bbox             borderWidth    text
  gutterWidth      editable         style          justification
  dropshadow       boxShadowSize
  shouldAutoTab    shouldAutoWrap
  ```

- DTKShape class

  ```
  backGroundColor  box              filled         foreGroundColor
  lineWidth        lineType         pattern        shapeStyle
  ```

- DTKSound class

  audiostream

- DTKButton class

  buttonStyle

- DTKPalette class

  colorMap

- DTKUnknown class

  nothing

Defining Customized Score Translator Classes

To specify how to translate a Director score into ScriptX objects, create a customized subclass of DTKScoreTranslator and define its prepareToTranslateScore and translateFrame methods. These methods are empty stubs, so you must define them completely for your subclass of score translator.

The class DTKScoreTranslator has an instance variable called outputStream. The intent of this instance variable is to point to an output stream associated with the score translator. The prepareToTranslateScore and translateFrame methods on the score translator, can, if desired, send information about the translated score to the output stream. (If the output stream is not defined, the information goes to the debug stream, which is the Listener window.)

Define the prepareToTranslateScore method to do any preliminary work necessary before translating the score, such as printing a message or creating support classes needed by the translation process. The prepareToTranslateScore method takes a single argument of *self*, the score translator object.

Define the translateFrame method to translate a frame in a Director score to ScriptX objects. You must decide how to represent the information about the frame in ScriptX. This method is the real workhorse for the score translation process.

Accessing Information about a Frame of the Director Score

For each frame in the score, the translateScore method passes three instances to the translateFrame method on the score translator:

- a DTKScoreFrame instance that contains information about the current frame.
- a DTKScoreFrame instance that contains information about the previous frame.
- an Array instance, known as the *changedArray* argument, that holds an array of 24 or 48 booleans (depending on which version of Director you are importing from). For each sprite channel, if a change has occurred between the previous and current frame, the corresponding value in the changed array is true, otherwise it is false.

The translateFrame method on the score translator can get information about the frame by accessing the instance variables on the DTKScoreFrame objects.

The instance variables on a DTKScoreFrame object are:

- absoluteFrameNumber

  Contains an integer indicating the frame number currently being processed.

- relativeFrameNumber

  Contains an integer indicating the relative position of the frame number currently being processed within the subset of frames to be translated.

- tempoChannel

  Contains a DTKTempoChannel object, that has tempo and waitForSound instance variables. The tempo instance variable indicates the tempo for the frame, and waitForSound indicates if this frame is waiting for a sound and if so, for which sound channel it is waiting for.

- transitionChannel

  Contains a DTKTransitionChannel object, that has name, direction and chunkiness instance variables that describe the transition occuring in this frame (if any).

- soundChannels

  Contains an array of two DTKSoundChannel objects, one for each sound channel in the Director title. Each DTKSoundChannel object has a castIndex instance variable, whose value indicates the position in the cast list of the sound cast member that is playing in this frame, if any.

- spriteChannels

Contains an array of 24 or 48 DTKSpriteChannel objects, depending on which version of Director the score was created in. Each DTKSpriteChannel object holds information about a single channel in the score. Each DTKSpriteChannel object has x, y, width, height, ink, thickness and castIndex instance variables, that describe the cast member, if any, in that channel in this frame.

DTKScoreFrame Objects and its Helpers are Recycled

The DTKScoreFrame, DTKTempoChannel, DTKTransitionChannel, DTKSoundChannel and DTKSpriteChannel objects are recycled as the score translation process continues. The translateFrame method should not keep references to the DTKScoreFrame object or any of the objects to which it points, since the references would become obsolete. Instead, write the translateFrame method to retrieve information from them, and implement the retrieved information as other objects in keeping with your own paradigm.

Comparing Two Frames to Check for Differences

When translating frames in a Director score to ScriptX, the important information often concerns the changes between one frame and the next. For example, has a sound started playing? Have any of the characters changed position? Have the inkmodes for any channels changed? Have any new characters appeared or any existing characters disappeared?

The translateFrame method on the score translator can compare the values of the instance variables on the prevFrame argument to those on the currentFrame argument.

You can also use the *changedArray* argument to check if there was a change between the current and prevous frame for each sprite channel.

The following code illustrates how to check if there was a tempo change between the previous and current frame.

```
method translateFrame {myScoreTrans} \
       self prevChannel currentChannel changedArray
(
   local currentTempo := currentFrame.tempoChannel.tempo
   local prevTempo:= prevFrame.tempoChannel.tempo
   if (currentTempo <> prevTempo)
   then
        format self.outputstream "\rThe tempo has changed to %* \r" \
          currentTempo @unadorned
   else
      format self.outputstream "\rThe tempo is still %* \r" \
          currentTempo @unadorned
-- rest of code for translateFrame
)
```

How the Score Translator Gets Information about Castmembers

In most cases, the cast translator and score translator need to work together to capture useful information about the castmembers used in a score. The cast translator returns objects that go into the castlist and the score translator can get objects from the castlist.

During the cast translation process, for each castmember in the original Director title (be it a bitmap, shape, text box, sound, video or button), the cast translator returns an object that goes into the ScriptX castlist.

While the cast translation process is underway, the castlist is maintained by the `castlist` instance variable on the relevant `DTK` instance. When the cast translation is finished, the `DTK` instance copies the list in its `castlist` instance variable over to the `castlist` instance variable on its score translator. Thus the score translator can get castmembers by referring to its own `castlist` instance variable.

Finding which Sound CastMember is Playing in a Channel

The `soundChannels` instance variable of a `DTKScoreFrame` object holds an array of two `DTKSoundChannel` objects, one for each sound channel in the score. To find the position in the castlist of the sound playing in a channel, get the value of the `castIndex` instance variable of the appropriate `DTKSoundChannel` object. If a sound is playing in the channel, the value of the `castIndex` instance variable will be 1 or 2. If a sound is not playing, it will be 0.

Finding which Sprite CastMember Appears in a Channel

The `spriteChannels` instance variable of a `DTKScoreFrame` object holds an array of `DTKSpriteChannel` objects, one for each sprite channel in the score. To find the position in the castlist of the visual castmember appearing in a sprite channel, get the value of the `castIndex` instance variable of the appropriate `DTKSpriteChannel` object. The value of the `castIndex` instance variable will be 1 to 48 if a sprite is in the channel, or 0 if no sprite is in the channel.

Example Code for a Score Translator

This section gives the code for an example `DTKScoreTranslator` subclass called `ScoreToInfoTranslator`. This class has a very simple mode of operation: it simply sends information about each frame in the score to its output stream. The purpose of this example is to illustrate how to get information out of the `DTKScoreFrame` objects. You would normally want to define the `translateFrame` method on your customized score translator class to actually do something useful with the retrieved information, rather than just print it as is done here.

In the following example, the score translator assumes that the cast translator has put an appropriate presenter in the `castList` array for each castmember. Each presenter contains information about the castmember it represents. (You can find this code in the `DTKsampl` folder on the CD.)

```
in Module myModule

-- Create the new class of Score Translator
class ScoreToInfoTranslator (DTKScoreTranslator)
end

method prepareToTranslateScore self {class ScoreToInfoTranslator} ->
(
   print "\nI am preparing to translate the score. Silence please.\n"
)

-- define the method that translates each frame
method translateFrame self {class ScoreToInfoTranslator} \
         prevFrame currentFrame changedArray ->
(
   local framenum := currentFrame.absoluteFrameNumber
```

```
    -- Print the frame number to the output stream
    format self.outputstream "\n\n\nProcessing frame number %* \n" \
       framenum @unadorned

    -- Translate data about the frame
    translateTempoDataToText self prevFrame currentFrame
    translateSoundDataToText self prevFrame currentFrame
    translateSpriteDataToText self prevFrame currentFrame changedArray
)

-------------------------------------------
-- USER-DEFINED METHODS TO DO TRANSLATING
-------------------------------------------

---------------------
-- Translate Tempo --
---------------------

method translateTempoDataToText self {class ScoreToInfoTranslator}\
       prevFrame currentFrame ->
(
    local currentTempo := currentFrame.tempochannel

    -- Print a message to the output stream that says whether the
    -- the tempo has changed since the previous frame, and
    -- what the current tempo is.
    if (currentFrame.tempoChannel.tempo <> prevFrame.tempoChannel.tempo)
    then
       format self.outputstream "\nThe tempo has changed to %* \n" \
          currentTempo.tempo @unadorned
    else
       format self.outputstream "\nThe tempo is still %* \n" \
          currentTempo.tempo @unadorned

    -- Lets see if we need to wait for any sounds.
    -- The value of the DTKTempoChannel's waitForSound iv is:
    -- 0 (don't wait for any sound),
    -- 1 (wait for the sound in channel 1) or
    -- 2 (wait for the sound in channel 2)
    if (currentFrame.tempoChannel.waitForSound <> 0) do
       format self.outputstream "We need to wait for the sound in channel %*
to finish \n" currentFrame.tempoChannel.waitForSound @unadorned
)

---------------------
-- Translate Sound --
---------------------

method translateSoundDataToText self {class ScoreToInfoTranslator} \
       prevFrame currentFrame ->
(
    -- A DTKScoreFrame object has a soundChannels iv that contains
    -- an array of two DTKSoundChannel objects, one for each sound channel

    -- For each sound channel, find which position the sound playing
    -- in that channel occupies in the cast list.
    -- Print sutiable messsages to the output stream.
    for i := 1 to 2 do
    (
       local oldIndex := prevFrame.soundChannels[i].castIndex
       local newIndex := currentFrame.soundChannels[i].castIndex

       -- If no sound is playing in this channel, castIndex will be 0
       if (newIndex <> 0)
       then
          format self.outputstream "Sound channel %1 is playing sound %2.\n"\
              #(i, self.castlist[newIndex].mediastream) \
              #(@unadorned, @unadorned)
       else
          format self.outputstream "Sound channel %* is currently silent.\n"\
              i @unadorned

       if newIndex <> oldIndex
       then
          format self.outputstream "This is a change from the previous frame.\n\
              1 @unadorned
       else
          format self.outputstream "This is the same as the previous frame.\n"\
              1 @unadorned
    ) -- close loop
)


-----------------------
-- Translate Sprites --
-----------------------

method translateSpriteDataToText self {class ScoreToInfoTranslator} \
       prevFrame currentFrame changedArray ->

(
    -- Get the cast list.
    theCastList := self.castlist

    -- For each sprite channel:
    for i := 1 to 48 do
    (
       -- Find which sprite channel we are currently focusing on.
       local theSpriteChannel := currentFrame.spriteChannels[i]
       local thePrevSpriteChannel := prevFrame.spriteChannels[i]

       -- Find which position the sprite that appears in
       -- that channel occupies in the cast list.
       local spritePosition := theSpriteChannel.castIndex
       local prevSpritePosition := thePrevSpriteChannel.castIndex

       -- Find the actual cast member object for the sprite in the channel
       thisCast := theCastList[spritePosition]
       prevCast := theCastList[prevSpritePosition]

       -- Print suitable messages to the output stream.
       if thisCast <> empty do
       (
          format self.outputstream "\n The castmember in channel %1 is %2.\n\
          Its name is %3. \n.
          It is at position %4 in the cast list. \n\
          This cast member is a %5. \n \
          Its x coordinate is %6. \n \
          Its y coordinate is %7.\n \
          Its inkmode is %8. \n \
          It is %9 high and %10 wide. \n\n" \
             #(i, thisCast, thisCast.name,
             spritePosition, (getClass thisCast), \
             theSpriteChannel.x,  theSpriteChannel.y, \
             thespriteChannel.ink, \
             theSpriteChannel.height, theSpriteChannel.width) \
             #(@unadorned, @unadorned, @unadorned, @unadorned, @unadorned,
             @unadorned, @unadorned, @unadorned, @unadorned)

          if changedArray[i] == false
          then
             print "No change in this sprite channel." self.outputstream
          else
          (
             -- Compare the ink mode of the current and previous frames
             if (thePrevSpriteChannel.ink <> theSpriteChannel.ink)
             do
                print "The ink mode has changed since the previous channel.\n" \
                   self.outputstream

             -- Comparing the state of the sprite in the
             -- current and previous frames

             if (thisCast <> prevCast)
             do
                print "The sprite has changed." self.outputstream

             if (thePrevSpriteChannel.x <> theSpriteChannel.x) or
                (thePrevSpriteChannel.y <> theSpriteChannel.y)
             do
                print "The sprite has changed position." self.outputstream

             if (thePrevSpriteChannel.width <> theSpriteChannel.width) or
                (thePrevSpriteChannel.height <> theSpriteChannel.height)
             do
                print "The sprite has changed size." self.outputstream
          ) -- closes if changedArray[i] == false

       ) -- closes if thisCast <> empty else

    ) -- closes for i := 1 to 48 do
 ) -- closes method
```

Importing a Director Animation into ScriptX

To import a Director animation into ScriptX, create an instance of the DTK class, an instance of your customized cast translator class and an instance of your customized score translator class.

Set the cast translator as the value of the DTK instance’s castTranslator instance variable, and set the score translator as the value of the scoreTranslator instance variable. To translate the cast only, set only the cast translator. To translate the score only, set only the score translator.

Start the translation by calling the translateDirector method on the DTK instance.

Creating a DTK Instance

To create a new DTK instance, call the new method on the DTK class, optionally specifying the directorFileName keyword argument as the Director file to be translated. If you do not specify the directorFileName keyword argument when creating a new DTK instance, you must set the value of the DTK instance's directorFileName instance variable before you set the DTK to work, otherwise it won't know what file to translate. The value of the directorFileName keyword argument or instance variable must be a sequence of strings that describe the pathname to the file, for example, #("HD", "Scores", "DancingDolphin").

Creating Cast Translator and Score Translator Instances

The new methods for the classes DTKCastTranslator and DTKScoreTranslator both accept an outputStream keyword argument that specifies the output stream associated with the translator. This output stream is set as the value of the outputStream instance variable of the cast or score translator.

If no outputStream keyword argument is supplied, the output stream defaults to debug, which means all data sent to the output stream appears in the ScriptX Listener window.

If the neither the prepareToTranslateCast method or the media translation methods on the cast translator send data to the output stream, there is no need to supply the outputStream keyword argument when creating the cast translator. If neither the prepareToTranslateScore or translateFrame methods on a score translator send data to the output stream, there is no need to supply the outputStream keyword argument when creating the score translator.

Start the Translation Rolling

To start a DTK instance translating the Director file pointed to by its directorFileName instance variable, call its translateDirector method, which takes an optional container keyword argument which specifies the storage container in which to store the imported media. The value for the container keyword argument can be a storage container, a title container or a library container.

The following example illustrates how to create a DTK instance and a cast translator and score translator for it, and how to start the DTK instance importing a Director animation. This example shows how to create and specify output streams for cast and score translators. This example creates instances of the CastToInfoTranslator and ScoreToInfoTranslator classes that were defined previously in this chapter. (You can find the sample code in the DTKsample folder on the CD.)

```
-- Load the DTK
open TitleContainer dir:theStartDir path:"dtk/dtk.sxl"

-- Create a module to work in
module myModule
   uses DTK
   uses ScriptX
end

in module myModule

-- load the files that define the CastToInfoTranslator
-- and ScoreToInfoTranslator classes
fileIn theScriptdir name:"myCtrans.sx"
fileIn theScriptdir name:"myStrans.sx"

-- make sure we are still in myModule
in module myModule

-- Create a new DTK instance
-- use the open file dialog box so the user can select a Director file
global locPanel := new OpenPanel
openFilePanel locPanel
dirFile := locPanel.filename

-- create an instance of DTK
myDTK := new DTK directorFileName:dirFile

-- if the files castData.txt and scoreData.txt already exist
-- then delete them
if isfile theScriptDir "castData.txt"
   do delete theScriptDir "castData.txt"
if isfile theScriptDir "scoreData.txt"
   do delete theScriptDir "scorData.txt"

-- create the files castData.txt and scoreData.txt
createFile theScriptDir "castData.txt" @text
createFile theScriptDir "scoreData.txt" @text

-- Create the output streams for the translators
global castStream := getStream theScriptDir "castData.txt" @readwrite
global scoreStream := getStream theScriptDir "scoreData.txt" @readwrite

-- Create a cast and score translator
global myst := new ScoreToInfoTranslator outputstream:scoreStream
global myct := new CastToInfoTranslator outputstream:castStream

-- tell the cast translator what classes to use when translating
-- shapes, bitmaps, text, and video
myct.shapeClass := CastTwoDShape
myct.textClass := CastTextPresenter
myct.bitmapClass := CastTwoDShape
myct.videoClass := CastMoviePlayer

-- Associate the Cast Translator with the DTK instance.
-- Associate the Score Translator with the DTK instance.

myDTK.castTranslator := myct
myDTK.scoreTranslator := myst

-- Set the DTK to work to import the data:
translateDirector myDTKtranslateDirector myDTK
```

Advanced Example

This section discusses another example. This example can be used to import scene-based Director title, that contain ScriptX tags as comments in lingo scripts.

This section gives an overview of the example and explains what it is trying to achieve, without delving into much code. All the files to recreate the example are on the CD, in the `DTKExmpl` folder.

The sample Director file provided in the `DTKExmpl` folder contains three scenes. One of the scenes contains three separate animations. In Director, these animations must all perform at the same tempo, whereas in ScriptX each animation can perform with a different tempo. Another of the scenes presents several musical instruments. In Director, only one instrument can play at a time, whereas in ScriptX multiple instruments can play simultaneously.

Preparing the Director Title

Create a prototype of your title in Director, and label the significant frames and channels. For example, label the start of each scene and the start of each distinct animation in a scene.

Write lingo scripts that contain "hints" for the ScriptX importer. These hints are written as comments in lingo. For example, for the frame that marks the beginning of scene 2 in the example Director file,the lingo script is:

```
--SXClass Scene
--SXEast Scene2
--SXsoundLooped yes
```

This script indicates that this frame is the start of a scene. When the frame is imported, a new `Scene` object will be created, whose east neighbor is `Scene2`. While the scene is open, the sound for the scene will loop continuously.

Another example is:

```
--SXClass Animation
--SXfirstFrame Eyelash
--SXlastFrame EyelashEnd
--SXfirstChannel 4
--SXlastChannel 7
--SXautoStart yes
--SXlooped yes
```

This script indicates that this frame is a start of an animation. The first frame is labelled `Eyelash`, and the last frame is labelled `EyelashEnd`. The first channel to participate in the animation is 4, and the last is 7. The animation should start automatically, and should loop continuously.

Another example is:

```
on mousedown --SXClass NavButton
  Lbtn "Scene1" --SXtoScene Scene1
end
--SXPressedPresenter LftBtnHi
```

This script indicates that this channel in this frame is a navigation button. When the button is pressed, the current scene closes and `Scene1` opens. When you press the button, the pressed presenter is the castmember named `LftBtnHi`.

How the Conversion Works

This section briefly discusses how the conversion works.

Custom Classes

The CustomClasses folder contains all the class definitons for the custom classes needed. For example, it contains definitions for the classes Scene, Animation, and NavButton amongst others. It also contains a file cci.mod, that loads all the custom classes and saves them to a library container, CustClass.sxl.

Several of the custom classes have specialized init or afterInit methods that take a keyword argument of lingo. The init or afterInit methods then use the findSXkey method (discussed in the section "Information for the Build Process") to search the lingo script for more information about the object being initialized.

Cast and Score Translators

The DTKSaver folder contains the definitions for the customized score and cast translators. It also contains a file that defines the function readMMfile, which opens a dialog box that lets you choose a Director file, then calls translateDirector on that file.

The folder contains a file DTKSaver.mak, which loads the class and function definitons and saves the classes and the readMMfile function to the library container DTKSaver.sxl.

Information for the Build Process

The folder Builder contains a file builder.sx that defines the functions needed for the conversion process, the main one being BuildATitle, and builds a library container, builder.sxl to contain the functions.

The function BuildATitle contains the information for converting castmembers to instances of ScriptX classes, and for parsing and using the information embedded in the lingo scripts.

One of the auxiliary functions used by BuildATitle is the function findSXKey which is defined in the library container globals.lib. This function prepends "--SX" to a string, and then searches a script for an occurence of the string and returns the word after the string.

For example, suppose the script is:

```
--SXClass Scene
--SXEast Scene2
--SXsoundLooped yes
```

In this case, the following command:

```
findKey(script "Class")
```

searches the given script for the first occurrence of "--SXClass" and returns the word after it in the script, which is Scene, which indicates the class to create an instance of.

The code below is an extract from BuildATitle. This extract illustrates how the importing process uses the information embedded in the lingo scripts to decide what to do.

In the code here, frameScript is bound to a lingo script. If the script contains the string "--SXclass", the variable newClassName is bound to the appropriate class. If the newClassName is Scene, print a header message, and create a new instance of the class Scene.

```
local frameScript := spriteChannels[1][@lingoscript]
if frameScript <> undefined do
(   local script := frameScript
    local newClassName := findSXKey(script, "Class")
    if newClassName = "Scene" do
    (   print "--------- SCENE ------------"
        local newScene := new Scene lingo:script boundary:(w.bbox)

        -- and so on
```

Converting the Title from Director to ScriptX

The file titleBuild.sx imports a Director title into ScriptX. This file requires that the library containers builder.sxl, CustClass.sxl, and dtkSaver.sxl reside in the same folder with it.

When you load the file titleBuild.sx, it calls the function readMMfile which opens a dialog box that lets you choose a Director title to import, and then calls translateDirector on the file, to start the translation process.

Further Modifying the Example

If you wish to extend the example to do other things, you need to decide on appropriate tags to put in lingo scripts. If you wish to use the function SXfindKey to search for tags, your tags must follow the format:

```
--SXsomething nextWord
```

For example:

```
--SXclass Dragon
--SXenemy SirGeorge
```

You also need to write your own cast and score translators, or copy and modify the cast and score translators that come with this example. The cast translator is defined in the folder DTKSaver in the file castSaver.sx, and the score translator is defined in the file ScoreSaver.sx.

You will need to define your own custom classes to implement the desired behavior for your title.

If you want to use the function buildTitle as the basis for a function that builds a title, copy it and modify it to suit your needs. You may need to add extra conditional clauses to take different tags or classes into account.

For example:

```
local frameScript := spriteChannels[1][@lingoscript]
if frameScript <> undefined do
(   local script := frameScript
    local newClassName:= findSXKey(script, "class")
    if newClassName = "Dragon" do
    (   print "--------- Dragon ------------"
local newDragon := new Dragon lingo:script boundary:(w.bbox)
-- and so on
```

CHAPTER

19

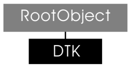
Director Translation Kit API

This chapter lists the classes in the Director Translation Kit (DTK). You can use the Director Translation Kit to build your own importers that convert either an entire title or specific components of a title from Macromedia Director to ScriptX.

The classes in the Director Translation Kit live in the DTK module, so you must work in the DTK module, or in a module that uses the DTK module, to be able to use them.

For information about how to use these classes to build a customized Director to ScriptX importer, see Chapter 18, "Using The Director Translation Kit."

DTK



Class type: Tool class (concrete)
Resides in: `dtk.sxl`
Inherits from: `RootObject`

The `DTK` class controls the process of importing a Director title into ScriptX. It has a `translateCast` method that translates the cast list and a `translateScore` method that translates the score. Its `translateDirector` method calls `translateCast` and `translateScore`.

Keywords for the New Method for DTKCastMemberToStencil

The `new` method for the `DTK` class takes the following keywords.

outputStream A stream associated with the translator which the translator can send data to.

container A storage container in which to store imported media.

toFrame The frame number from which to start importing.

fromFrame The frame number up to which to import.

Instance Variables

backgroundColorIndex

self.`backgroundColorIndex` (read-write) `Number`

Specifies the index into the current color map that the background fill color should be. Currently the color map is always the system color map.

castList

self.`castList` (read-write) `Array`

An array of ScriptX objects that represent the castmembers imported from a Macromedia Director score. The members of the array are text sprites, 2D sprites, video sprites and audio streams. All cast members that are not successfully translated are represented by `undefined` objects.

This array is updated as the cast translation occurs. Before the cast translation begins, its value is an empty array.

castTranslator

self.`castTranslator` (read-write) `DTKCastTranslator`

The `DTKCastTranslator` used to convert a Macromedia Director castlist to ScriptX objects.

defaultFrameRate

self.`defaultFrameRate` (read-write) `Number`

The default frame rate for a Macromedia Director movie score. The score translator uses this instance variable.

directorFilename

self.`directorFilename` (read-write) `Number`

The file name of the Director file to be imported.

labelList

self.labelList (read-write) Array

A list of ScriptX Marker objects that represent the labels in a Macromedia Director movie score.

numberOfFrames

self.numberOfFrames (read-write) Number

The number of frames to be imported.

numberOfCastMembers

self.numberOfMembers (read-write) Number

The number of cast members in the array in the castList instance variable.

scoreTranslator

self.scoreTranslator (read-write) DTKScoreTranslator

The DTKScoreTranslator used to convert a Macromedia Director score to ScriptX objects.

stageRect

self.stageRect (read-write) Rect

The rect object to use as the boundary for the space which will be used as the stage for the imported animation.

Instance Methods

translateCast

translateCast *self* ⇨ undefined

Converts a Director cast list into ScriptX objects.

translateDirector

translateDirector *self* ⇨undefined

self	DTK object

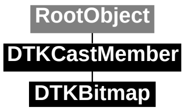
Converts a Macromedia Director animation to ScriptX objects, using the DTK *self*'s cast translator and score translator, if any.

translateScore

translateScore *self* ⇨ self

Converts a Director score into ScriptX objects.

DTKBitmap



Class type: Tool class (concrete)
Resides in: `dtk.sxl`
Inherits from: `DTKCastMember`

A `DTKBitmap` instance holds information about a Director bitmap castmember.

Instance Variables

Inherited from `DTKCastMember`:

`castFileName` `castFileType` `castFullPath`
`castName` `castNumber` `container`
`lingoScript` `linked`

The following instance variables are defined in `DTKBitmap`:

bitmap

self`.bitmap` (read-write) `Bitmap`

Specifies a `Bitmap` object that corresponds to the bitmap used by a Director bitmap castmember.

bBox

self`.bBox` (read-write) `Rect`

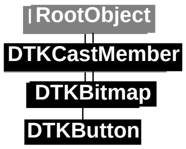
Specifies a ScriptX `Rect` object that is the bounding rectangle of the bitmap.

registrationPt

self`.registrationPt` (read-write) `Point`

A `Point` object that represents the registration point for a Macromedia Director bitmap graphic.

DTKButton



Class type: Tool class (concrete)
Resides in: `dtk.sxl`
Inherits from: `DTKText`

A `DTKButton` class holds information about a Director button castmember.

Instance Variables

Inherited from `DTKCastMember`:

`castFileName` `castFileType` `castFullPath`
`castName` `castNumber` `container`
`lingoScript` `linked`

Inherited from `DTKText`:

`backgroundColor` `bBox` `borderWidth`
`text`

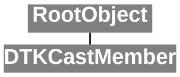
The following instance variables are defined in `DTKButton`:

buttonStyle

self.`buttonStyle` (read-write) `NameClass`

The style of the button being converted—either `@pushbutton`, `@radioButton`, or `@checkbox`.

DTKCastMember



Class type: Tool class (abstract)
Resides in: `dtk.sxl`
Inherits from: `RootObject`

The `DTKCastMember` class is the base class for all classes that represent castmembers imported from Macromedia Director.

Instance Variables

castFileName

self.`castFileName` (read-write) `String`

The name of a linked file. If the value of the instance variable `linked` is `false`, the `castFileName` instance variable is an empty string.

castFileType

self.`castFileType` (read-write) `String`

The Macintosh file type of a linked file, which is a `String` object. If the value of the instance variable `linked` is `false`, the `castFileType` instance variable is an empty string.

castFullPath

self.`castFullPath` (read-write) `String`

The fully qualified path to a linked file. If the value of the instance variable `linked` is `false`, the `castFullPath` instance variable is an empty string.

castName

self.`castName` (read-write) `String`

The name of the cast being imported from a Macromedia Director cast list.

castNumber

self.`castNumber` (read-write) `String`

The number of the cast being imported from a Macromedia Director cast list.

container

self.`container` (read-write) `String`

The container in which to store the media for the cast member .

lingoScript

self.`lingoScript` (read-write) `String`

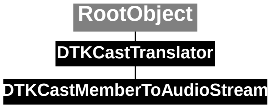
The unconverted Lingo script for the cast member.

linked

self.`linked` (read-write) `Boolean`

If the castmember media is linked to a file on a disk, the value of this instance variable is `true`; if not, `false`.

DTKCastMemberToAudioStream



Class type: Tool class (concrete)
Resides in: dtk.sxl
Inherits from: DTKCastTranslator

The DTKCastMemberToAudioStream class is a specialized subclass of the DTKCastTranslator class that converts a Macromedia Director sound castmember to an AudioStream object in ScriptX.

Instance Variables

Inherited from DTKCastTranslator:

```
dtk                    container              outputStream
warnings
```

Instance Methods

Inherited from DTKCastTranslator:

```
setDTK                 translateBitmap        translateButton
translateShape         translateSound         translateText
translateUnknown       translateVideo
```

The inherited definitions for translateButton, translateShape, translateBitmap, translateText and and translateUnknown methods all simply return undefined.

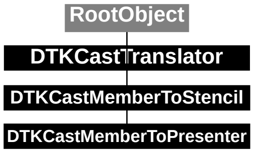
The following instance methods are defined in DTKCastMemberToAudioStream:

translateSound

translateSound *self soundCastMember* ⇨ AudioStream

Translates a Macromedia Director sound castmember to an AudioStream object in ScriptX.

DTKCastMemberToPresenter



Class type: Tool class (concrete)
Resides in: `dtk.sxl`
Inherits from: `DTKCastMemberToStencil`

The `DTKCastMemberToPresenter` class is a specialized subclass of the `DTKCastTranslator` class that converts a Macromedia Director shape, bitmap, text or video castmember to an appropriate presenter object in ScriptX.

Keywords for New for DTKCastMemberToPresenter

The `new` method for the `DTKCastMemberToPresenter` class takes `outputStream`, `shapeClass`, `bitmapClass` and `textClass` keywords.

outputStream A stream associated with the translator which the translator can send data to.

shapeClass The class of `Presenter` that the `translateShape` method will translate shape castmembers to. The default is `TwoDShape`.

bitmapClass The class of `Presenter` that the `translateBitmap` method will translate bitmap castmembers to. The default is `TwoDShape`.

textClass The class of `Presenter` that the `translateText` method will translate text castmembers to. The default is `TextPresenter`.

videoClass The class of Presenter that the translateVideo method will translate video castmembers to. The default is InterleavedMoviePlayer.

Instance Variables

Inherited from `DTKCastTranslator`:

`dtk` `container` `outputStream`
`warnings`

The following instance variables are defined in `DTKCastMemberToPresenter`:

bitmapClass

self`.bitmapClass` (read-write) `Class`

Specifies which class to use as the parent for the object created when the `translateBitmap` method on the translator *self* translates a bitmap castmember to a ScriptX object.

shapeClass

self`.shapeClass` (read-write) `Class`

Specifies which class to use as the parent for the object created when the `translateShape` method on the translator *self* translates a shape castmember to a ScriptX object.

textClass

self.`textClass` (read-write) `Class`

Specifies which class to use as the parent for the object created when the `translateText` method on the translator *self* translates a text castmember to a ScriptX object.

videoClass

self.`videoClass` (read-write) `Class`

Specifies which class to use as the parent for the object created when the `translateVideo` method on the translator *self* translates a video castmember to a ScriptX object.

Instance Methods

Inherited from `DTKCastTranslator`:

```
setDTK               translateBitmap      translateButton
translateShape       translateSound       translateText
translateUnknown     translateVideo
```

The inherited definitions for `translateButton`, `translateSound` and `translateUnknown` methods all simply return undefined.

The following instance methods are defined in `DTKCastMemberToPresenter`:

translateBitmap

`translateBitmap` *self bitmapCastmember* ⇨ `TwoDShape`

Converts a Macromedia Director bitmap castmember to a `TwoDShape` object in ScriptX.

translatePalette

`translatePalette` *self paletteCastmember* ⇨ `ColorMap`

Converts a palette castmember to a `ColorMap` object in ScriptX.

translateShape

`translateShape` *self shapeCastmember* ⇨ `TwoDShape`

Converts a Macromedia Director shape castmember to a `TwoDShape` object in ScriptX. This method generates a brush for the fill and stroke of the presenter, whose color and pattern are retrieved from the `DTKShape` object.

translateText

`translateText` *self textCastmember* ⇨ `TextPresenter`

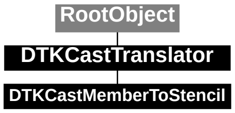
Converts a Macromedia Director text castmember to a `TextPresenter` object in ScriptX.

translateVideo

`translateVideo` *self videoCastmember* ⇨ `InterleavedMoviePlayer`

Converts a Macromedia Director video castmember to an `InterleavedMoviePlayer` object in ScriptX.

DTKCastMemberToStencil



Class type: Tool class (concrete)
Resides in: `dtk.sxl`
Inherits from: `DTKCastTranslator`

The `DTKCastMemberToStencil` class is a specialized subclass of the `DTKCastTranslator` class that converts a Macromedia Director graphic castmember to a `Stencil` object in ScriptX.

Keywords for the New Method for DTKCastMemberToStencil

The `new` method for the `DTKCastMemberToStencil` class takes an `outputStream` keyword.

outputStream A stream associated with the translator which the translator can send data to.

Instance Variables

Inherited from `DTKCastTranslator`:

`dtk` `container` `outputStream`
`warnings`

Instance Methods

Inherited from `DTKCastTranslator`:

`setDTK` `translateBitmap` `translateButton`
`translateShape` `translateSound` `translateText`
`translateUnknown` `translateVideo`

The inherited definitions for `translateButton`, `translateSound`, `translateVideo`, and `translateUnknown` methods all simply return undefined.

The following instance methods are defined in `DTKCastMemberToPresenter`:

translateBitmap

`translateBitmap` *self bitmapCastmember* ⇨ `Bitmap`

Converts a Macromedia Director bitmap castmember to a `Bitmap` object in ScriptX.

translatePalette

`translatePalette` *self paletteCastmember* ⇨ `ColorMap`

Converts a palette castmember to a `ColorMap` object in ScriptX.

translateShape

`translateShape` *self bitmapCastmember* ⇨ `Stencil`

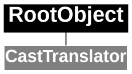
Converts a Macromedia Director shape castmember—such as a `Rect`, `RoundRect`, `Line`, or `Oval`—to an appropriate `Stencil` object in ScriptX.

translateText

`translateText` *self textCastmember* ⇨ `Text`

Converts a text castmember to a `Text` object in ScriptX.

DTKCastTranslator



Class type: Tool class (abstract)
Resides in: `dtk.sxl`
Inherits from: `RootObject`

The `DTKCastTranslator` class is the base class for all cast translators. A cast translator converts Macromedia Director castmembers into ScriptX objects.

Instance Variables

container

self`.container` (read-write) `StorageContainer`

The `StorageContainer` object in which the cast translator *self* can store media used by translated cast members.

dtk

self`.dtk` (read-write) `DTK`

The `DTK` object that will work with the translator *self* to translate the Director castlist.

outputStream

self`.outputStream` (read-write) `Stream`

A stream associated with the translator *self*. The translator can write data to this stream during cast translation if desired.

warnings

self`.warnings` (read-write) `Boolean`

When `true`, specifies that the warnings are to be printed; when `false`, warnings are not to be printed. Default is `true`.

Instance Methods

prepareToTranslateCast

`prepareToTranslateCast` *self* ⇨ `undefined`

Does whatever work is necessary before translating the castlist. By default, this method is undefined. Define it if you want to on your customized cast translator class.

translateBitMap

`translateBitMap` *self bitmapCastmember* ⇨ `undefined`

Translates a bitmap castmember to a ScriptX object. The default object created is `undefined`. Subclasses should override this method to return a meaningful object.

translateButton

`translateButton` *self buttonCastmember* ⇨ `undefined`

Translates a button castmember to a ScriptX object. The default object created is `undefined`. Subclasses should override this method to return a meaningful object.

translatePalette

`translatePalette` *self paletteCastmember* ⇨ `undefined`

Translates a palette castmember to a ScriptX object. The default object created is `undefined`. Subclasses should override this method to return a meaningful object.

translateShape

`translateShape` *self shapeCastmember* ⇨ `undefined`

Translates a shape castmember to a ScriptX object. The default object created is `undefined`. Subclasses should override this method to return a meaningful object.

translateSound

`translateSound` *self soundCastmember* ⇨ `undefined`

Translates a sound castmember to a ScriptX object. The default object created is `undefined`. Subclasses should override this method to return a meaningful object.

translateText

`translateText` *self textCastmember* ⇨ `undefined`

Translates a text castmember to a ScriptX object. The default object created is `undefined`. Subclasses should override this method to return a meaningful object.

translateUnknown

`translateUnknown` *self castmember* ⇨ `undefined`

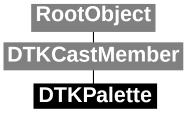
Translates a castmember other than a button, sound, video, text, bitmap or shape castmember. The default object created is `undefined`.

translateVideo

`translateUnknown` *self videoCastmember* ⇨ `undefined`

Translates a video castmember to a ScriptX object. The default object created is `undefined`. Subclasses should override this method to return a meaningful object.

DTKPalette



Class type: Tool class (concrete)
Resides in: `dtk.sxl`
Inherits from: `DTKCastMember`

A `DTKPalette` instance holds information about a Director palette castmember.

Instance Variables

Inherited from `DTKCastMember`:

`castFileName` `castFileType` `castFullPath`
`castName` `castNumber` `container`
`lingoScript` `linked`

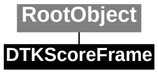
The following instance variables are defined in `DTKPalette`:

colorMap

self`.colorMap` (read-write) `ColorMap`

A ScriptX `ColorMap` object that represents the palette of the cast member.

DTKScoreFrame



Class type: Tool class (concrete)
Resides in: `dtk.sxl`
Inherits from: `RootObject`

A `DTKScoreFrame` instance holds information about a frame in a Director score.

Instance Variables

absoluteFrameNumber

self.`absoluteFrameNumber` (read-write) `Number`

The number of the frame in the Macromedia Director movie score that the `DTKScoreFrame` object represents.

relativeFrameNumber

self.`relativeFrameNumber` (read-write) `Number`

The number of the frame in the Macromedia Director movie score that the `DTKScoreFrame` object represents, in relation to the point at which the translation began.

scriptChannels

self.`scriptChannels` (read-write) `Array`

An array of scripts for each channel for this frame in the score.

soundChannels

self.`soundChannels` (read-write) `Array`

An array that consists of two `DTKSoundChannel` objects for the frame.

spriteChannels

self.`spriteChannels` (read-write) `Array`

An array that consists of 24 or 48 `DTKSpriteChannel` objects for the frame, depending on the version of Director that the imported file was created in.

tempoChannel

self.`tempoChannel` (read-write) `DTKTempoChannel`

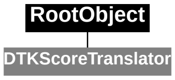
The `DTKTempoChannel` object for the frame.

transitionChannel

self.`transitionChannel` (read-write) `DTKTransitionChannel`

The `DTKTransitionChannel` object for the frame.

DTKScoreTranslator



Class type: Tool class (abstract)
Resides in: `dtk.sxl`
Inherits from: `RootObject`

The `DTKScoreTranslator` class is the base class for all score translators. A score translator converts a Macromedia Director score to ScriptX.

Keywords for the New Method for DTKCastMemberToStencil

The `new` method for the `DTKScoreTranslator` class takes the following keywords.

outputStream A stream associated with the translator which the translator can send data to.

Instance Variables

castList

self`.castList` (read-write) `Array`

An array of the sprites and audio streams that corrspond to the castmembers in the imported Director cast. If the cast has not been imported, the value of this instance variable is undefined.

container

self`.container` (read-write) `StorageContainer`

The `StorageContainer` object in which to store the media for the imported score.

dtk

self`.dtk` (read-write) `DTK`

The DTK object that is controlling the `DTKScoreTranslator` object.

outputStream

self`.outputStream` (read-write) `Stream`

The stream that the translator *self* can send data to, if desired.

Instance Methods

prepareToTranslateScore

`prepareToTranslateScore` *self* ⇨ `self`

This method is called before the score translator *self* starts translating the score. By default, it does nothing.

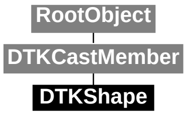
translateFrame

`translateFrame` *self currentFrame previousFrame changedArray* ⇨ `self`

self	`DTKScoreFrame` object
currentFrame	`DTKScoreFrame` object
previousFrame	`DTKScoreFrame` object
changedArray	`Array` object

When converting a Macromedia Director movie score, the `DTK` calls this method for each frame of the movie score.

DTKShape



Class type: Tool class (concrete)
Resides in: dtk.sxl
Inherits from: DTKCastMember

A DTKShape instance holds information about a Director shape castmember.

Instance Variables

Inherited from DTKCastMember:

castFileName castFileType castFullPath
castName castNumber container
lingoScript linked

The following instance variables are defined in DTKShape:

backgroundColor

self.backgroundColor (read-write) RGBColor

A ScriptX RGBColor object that represents the background color of the shape.

bBox

self.bBox (read-write) Rect

A ScriptX Rect object that is the bounding rectangle of the shape.

filled

self.filled (read-write) Boolean

If the shape is filled, the value of this instance variable is true.

foregroundColor

self.foregroundColor (read-write) RGBColor

A ScriptX RGBColor object that represents the foreground color of the shape.

lineWidth

self.lineWidth (read-write) Number

The thickness of the line that constitutes the border of the shape.

lineType

self.lineType (read-write) NameClass

For lines, specifies which way the line runs. If *self* is a line that runs from top left to bottom right the value is @criss. If *self* is a line that runs from bottom left to top right, the value is @cross. If *self* is not a line, the value is undefined.

pattern

self.pattern (read-write) Bitmap

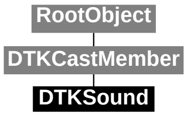
Specifies a bitmap for the one bit pattern for the shape *self*. (It does not return 8 bit patterns, or tiles, yet.)

shapeStyle

self`.shapeStyle`	(read-write)	`Symbol`

The type of the stencil—either `@rect`, `@roundRect`, `@line`, or `@oval`.

DTKSound



Class type: Tool class (concrete)
Resides in: `dtk.sxl`
Inherits from: `DTKCastMember`

A `DTKSound` instance holds information about a Director sound castmember.

Instance Variables

Inherited from `DTKCastMember`:

`castFileName`
`castFileType`
`castFullPath`
`castName`
`castNumber`
`container`
`lingoScript`
`linked`

The following instance variables are defined in `DTKSound`:

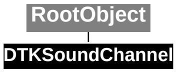
Instance Variables

audioStream

self`.audioStream` (read-write) `AudioStream`

The `AudioStream` object that is associated with the sound.

DTKSoundChannel



Class type:	Tool class (concrete)
Resides in:	`dtk.sxl`
Inherits from:	`RootObject`

The `DTKSoundChannel` class represents a sound channel for a frame of a Macromedia Director movie score.

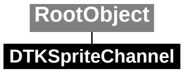
Instance Variables

castIndex

self`.castIndex` (read-write) `Number`

An index into the array of castmembers that the `DTKCastTranslator` generates. This array resides in the `castList` instance variable on the `DTK` class.

DTKSpriteChannel



Class type: Tool class (concrete)
Resides in: `dtk.sxl`
Inherits from: `RootObject`

The `DTKSpriteChannel` class represents a sprite channel for a frame of a Macromedia Director movie score.

Instance Variables

absoluteChannelNumber

self`.absoluteChannelNumber` (read-write) `Number`

The number of the Macromedia Director sprite channel that the `DTKSpriteChannel` object is converting.

castIndex

self`.castIndex` (read-write) `Number`

Specifies which position in the castlist is occupied by the sprite currently appearing in the channel *self*.

height

self`.height` (read-write) `Number`

The height of the sprite in the channel *self*.

ink

self`.ink` (read-write) `NameClass`

Specifies the ink mode for the channel *self*. The value is `@Copy`, `@Matte`, or `@Invisible`.

lingoScript

self`.lingoScript` (read-write) `NameClass`

The unconverted Lingo script for the channel *self*.

relativeChannelNumber

self`.relativeChannelNumber` (read-write) `Number`

The number of the Macromedia Director sprite channel that the `DTKSpriteChannel` object is currently converting, in relation to the channel that it first converted.

thickness

self`.thickness` (read-write) `Number`

Specifies the thickness of the border of the sprite in channel *self*.

useRectFromChannel

self`.useRectFromChannel` (read-write) `Boolean`

Specifies ... in channel *self*.

width

self`.width`	(read-write)	`Number`

Specifies the width of the sprite in channel *self*.

x

self`.x`	(read-write)	`Number`

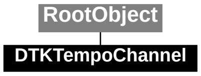
Specifies the `x` coordinate of the location on the stage of the sprite in channel *self*.

y

self`.y`	(read-write)	`Number`

Secifies the `y` coordinate of the location on the stage of the sprite in channel *self*.

DTKTempoChannel



Class type: Tool class (concrete)
Resides in: `dtk.sxl`
Inherits from: `RootObject`

The `DTKTempoChannel` class represents a tempo channel for a frame of a Macromedia Director movie score.

Instance Variables

tempo

self`.tempo` (read-write) `Number`

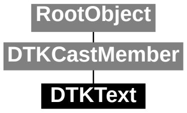
The tempo for a frame of the Macromedia Director movie score being imported.

waitForSound

self`.waitForSound` (read-write) `Integer`

Indicates which sound channel to wait for, if any.

DTKText



Class type: Tool class (concrete)
Resides in: `dtk.sxl`
Inherits from: `DTKCastMember`

A `DTKText` instance holds information about a Director text castmember.

Instance Variables

Inherited from `DTKCastMember`:

```
castFileName        castFileType        castFullPath
castName            castNumber          container
lingoScript         linked
```

The following instance variables are defined in `DTKText`:

backgroundColor

self.`backColor` (read-write) `RGBColor`

A ScriptX `RGBColor` object that represents the background color of the text.

bbox

self.`bbox` (read-write) `Rect`

A ScriptX `Rect` object that is the bounding rectangle of the text.

borderWidth

self.`borderWidth` (read-write) `Number`

Specifies the width of the border for the Director text castmember that has been converted to the `DTKText` object *self*.

boxShadowSize

self.`boxShadowSize` (read-write) `Number`

Specifies the size of the shadow for the box for the Director text castmember that has been converted to the `DTKText` object *self*.

dropShadowSize

self.`dropShadowSize` (read-write) `Number`

Specifies the size of the drop shadow for the Director text castmember that has been converted to the `DTKText` object *self*.

editable

self.`editable` (read-write) `Boolean`

Specifies whether the text should be editable or not.

gutterWidth

self.`gutterWidth` (read-write) `Number`

Specifies the width of the gutter for the Director text castmember that has been converted to the `DTKText` object *self*.

justification

self.`justification` (read-write) `NameClass`

Specifies the justification for the Director text castmember that has been converted to the `DTKText` object *self*. Values are `@flushleft` (possibley `@left`), `@flushright` (possibly `@right`), and `@center`.

shouldAutoTab

self.`shouldAutoTab` (read-write) `Boolean`

Specifies whether the text should have autotabs or not.

shouldWrap

self.`shouldWrap` (read-write) `Boolean`

Specifies whether the text should wrap or not.

style

self.`style` (read-write) `NameClass`

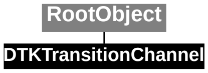
Specifies the style of the text for the Director text castmember that has been converted to the `DTKText` object *self*. Possible values are `@adjustToFit`, `@scrolling`, `@fixedSize`, and `@lineToWidth.`

text

self.`text` (read-write) `Text`

Specifies the text displayed by the Director text castmember that has been converted to the `DTKText` object *self*. The text retains the attributes such as justification and typestyle that it had in the original Director cast member.

DTKTransitionChannel



Class type: Tool class (concrete)
Resides in: `dtk.sxl`
Inherits from: `RootObject`

The `DTKTransitionChannel` class represents a transition channel for a frame of a Macromedia Director movie score.

Instance Variables

chunkiness

self`.chunkiness` (read-write) `Number`

A number from the Macromedia Director document. You can define its meaning in ScriptX.

direction

self`.direction` (read-write) `NameClass`

The direction in which the transition *self* moves, which can be `@up`, `@down`, `@left`, or `@right`.

duration

self`.duration` (read-write) `Number`

The number of frames that the transition *self* takes to complete.

name

self`.name` (read-write) `NameClass`

The type of the transition *self*, which can be `@wipe`, `@barndoor`, `@iris`, or `@slide`.

DTKUnknown

Class type: Tool class (concrete)
Resides in: `dtk.sxl`
Inherits from: `DTKCastMember`

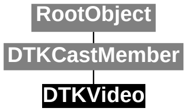
The `DTKUnknown` class is used to represent Director castmembers that cannot be translated into ScriptX objects, because the cast type is not recognised or not supported by the Director Translation Kit.

Instance Variables

Inherited from `DTKCastMember`:

```
castFileName          castFileType          castFullPath
castName              castNumber            container
lingoScript           linked
```

DTKVideo



Class type: Tool class (concrete)
Resides in: dtk.sxl
Inherits from: DTKCastMember

A DTKVideo instance holds information about a Director video castmember.

Instance Variables

Inherited from DTKCastMember:

castFileName castFileType castFullPath
castName castNumber container
lingoScript linked

The following instance variables are defined in DTKText:

directToStage

self.directToStage (read-write) Boolean

Specifies whether the direct to stage attribute is set to true or not.

moviePlayer

self.moviePlayer (read-write) InterleavedMoviePlayer

A ScriptX InterleavedMoviePlayer object that can be used to play the imported movie.

pausedAtStart

self.pausedAtStart (read-write) Boolean

Specifies whether the movie is paused at the start or not.

showController

self.showController (read-write) Boolean

Specifies whether to show the controller or not.

useSound

self.useSound (read-write) Boolean

Specifies whether the movie uses sound or not.

PART FOUR

Extending ScriptX

CHAPTER

Extending ScriptX

20

Most of the code in a ScriptX title or tool is compiled into the cross-platform bytecode format which can be executed by any ScriptX run-time engine. For the current release, the supported platforms for ScriptX are OS/2, the Apple Macintosh (68040 and PowerPC), and Microsoft Windows-based systems.

Occasionally, you may need to write code that is either platform-specific, or that requires resources or performance levels not available from the scripter. The Kaleida Media Player allows you to include additional routines in your projects, created in C. ScriptX allows these external code libraries (called *extensions*) to be loaded and run on the platform for which they are compiled.

Using extensions, you can

- use special-purpose routines already in existence.
- generate high-performance subroutines.
- access platform features not necessarily exposed to the scripting language.

The disadvantage to extensions is that compiled code can be used on only a single platform. You must compile separate versions of your code for each machine or operating system on which you expect to run it. Once the code is written and compiled, you can load this external code and call your routines from within the run-time environment.

When creating extensions you should also create a scripted version of any extension functions, if possible. This allows the functions to be performed on any platform for which the C version of the extension is not available.

To support the execution of platform-specific extensions, the ScriptX run-time engine includes a `Loader` class that is capable of loading extensions and binding them into the run-time engine. In addition, the Loader component includes a `MemoryObject` class that facilitates the exchange of data between extensions and the run-time engine, and provides static long-term storage for loadable extensions. The classes in the Loader component are documented in the *ScriptX Class Reference*.

This chapter is organized in two sections. The first part, “Using the Loader to Run Loadable Extensions,” describes how to load C-based extensions into ScriptX. The second part, “Writing Loadable Extensions” on page 322, describes how to write both scripter code and loadable code to allow for communication at run time between ScriptX and the loadable extension.

Using the Loader to Run Loadable Extensions

The Loader component provides a mechanism for binding platform-specific code to a particular run-time engine. A release disk can contain loadable code for several platforms. The loader takes care of loading the appropriate units for the current platform.

To organize and maintain separate code for each platform, the loader uses the hierarchical directory structure of the platform’s file system. A *loadable group* is a collection of compiled binary code files known as *loadable units*. The name of the loadable group is the name of the directory (or folder) containing the loadable units. Within the loadable group directory is another series of directories, one for each platform for which code exists. Within these platform directories are the loadable units.

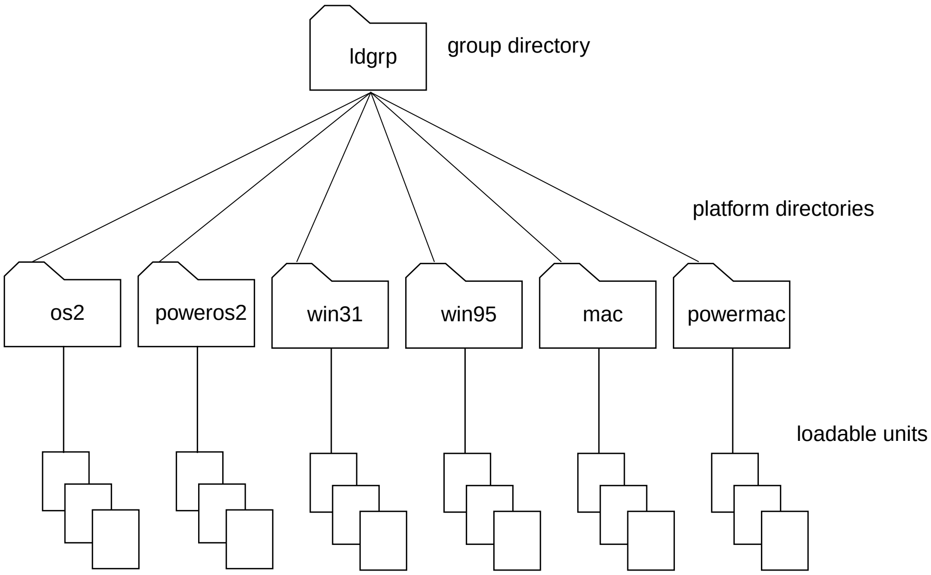


Figure 20-1: Hierarchical directory structure of a loadable group.

Each platform directory contains a text file named `group`, which describes the loadable units for that platform. The group file, described below, maps symbolic names of loadable units to specific files (which must have file names that are legal for the platform).

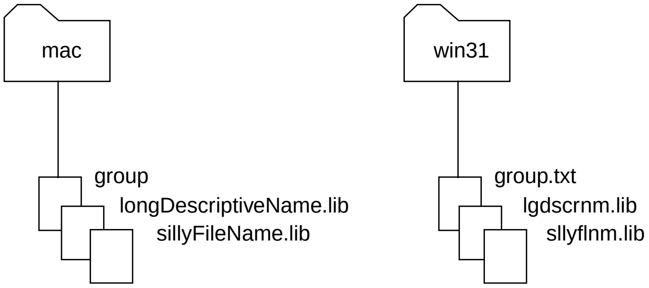


Figure 20-2: Contents of a loadable group's platform directory.

The Group File

In Figure 20-1, the loadable group is named `ldgrp`. When given the command to begin processing the group, the loader searches the directory corresponding to the current platform and opens the file named `group`. This file is a text database describing the contents of the loadable group and the attributes of each of the loadable units in the directory.

The files that make up a group are simply a logical collection of files and are not physically combined in any way. The units in a group are not necessarily loaded at the same time, although all loadable units are brought into the system when the `group` file is read.

For each loadable unit, the group file contains seven lines of text. These lines of text define attributes of a loadable unit and must appear in the order described below. Comment lines, indicated by a hash sign (#) in the first position in the line, can precede each unit description. Blank lines between unit descriptions are ignored. The following list summarizes each of these seven lines of text:

1. Loadable unit name.

   The loadable unit name can be any string of non-blank ASCII characters.

2. File name containing the loadable unit.

   The file name must be a legal file name for the platform. It identifies one of the binary files in the platform directory.

3. Symbolic name of the loadable unit's entry point and exit point.

   The entry point name is the name of an executable function inside the binary file that is called as soon as the unit is loaded. The exit point is an optional symbolic name, separated from the entry point by white space. The exit point function is called when the loadable unit is relinquished.

4. System version number.

   The system version number indicates compatibility with ScriptX versions. This number is usually 0, indicating that the unit is compatible with any ScriptX version. If the system version number is not 0, then the unit cannot be loaded unless the number matches the version number of the currently executing Kaleida Media Player.

5. Loadable unit version number.

   The loadable unit version number identifies the unit. If the loader encounters more than one loadable unit of the same name, it loads the one with the highest version number.

6. Loadable unit type.

   The loadable unit type must be one of two strings, either LoaderTypeLinkable or LoaderTypeEphemeral. (For more information about the unit type, see "Loadable Units" later in this chapter.)

7. Load list flag.

   The load list flag must be set to either true or false. The load list flag is used if the loader has been invoked using the process method. The process method tells the loader to walk through the entire group file and load every unit that has the value in its load list flag line set to true.

The following example illustrates a group file containing three loadable units named registration, JPEGImporter, and GIFImporter. Note that only the first unit has its load list flag set to true.

```
#Registers other units with the system
registration
register.lib
startUp   shutDown
0
1
LoaderTypeEphemeral
true

#Contains JPEG importer
JPEGImporter
jpeg.lib
initClass
```

```
0
1
LoaderTypeLinkable
false

#Contains the GIF importer
GIFImporter
gif.lib
initClass
0
1
LoaderTypeLinkable
false
```

Loadable Units

Each loadable unit corresponds to a compiled code extension. Every loadable unit must contain at least one function that is the designated entry point for the unit. When the loader successfully brings the unit into memory, the entry point function is invoked. The entry point function typically performs initialization chores that are necessary for successful operation of other routines in the unit.

If the purpose of the loadable unit is to perform a single action at load time, such as defining a set of properties, initializing a hardware component of the system, or something similar, the unit should have the loader unit type `LoaderTypeEphemeral`. (The loader unit type is declared in the sixth line of the unit description.) When an ephemeral unit is brought into memory, its entry point function is invoked. The unit is then released, allowing the garbage collector to remove it from memory.

Loadable units that contain class definitions or functions that are to be called after the unit is loaded must have a loader unit type `LoaderTypeLinkable`. These units are bound into the run-time engine and act as if they were part of the substrate. Linkable units can be released from memory after they have been used. This allows the system to dynamically load and discard code that is not used frequently, such as import or export routines.

Invoking the Loader

You create an active instance of the loader by calling the `new` method on the `Loader` class. You can then invoke methods on the instance of `Loader` to process loadable groups and loadable units. The loader creates instances of `LoadableGroup`, `LoadableUnit`, and `LoadableUnitId` to represent loadable groups, loadable units, and loadable units that are currently in memory. (For more information on the `Loader` class, see the *ScriptX Class Reference.*)

Processing Complete Groups

The simplest way to invoke the loader is to call the `process` method. The `process` method loads every unit in the `group` file that has its load list flag set to `true`. An example using the `process` method might look like this:

```
myLoader := new Loader
myLoadableIDList := process myLoader "ldgroup"
```

These two expressions create a loader instance and tell it to process the group in the directory ldgroup. The loader returns a list of LoadableUnitID objects for units that are loaded into memory. The loader expects to find the directory ldgroup in the directory from which ScriptX was launched.

Since ephemeral units are discarded immediately after loading, process never returns a LoadableUnitID object for an ephemeral unit. In addition, you do not get LoadableUnitID objects for units that have their load list flag set to false, since these units are not brought into memory at all.

Note – If you process a loadable group more than once, the loader will make sure that units marked LoaderTypeLinkable are not brought into memory more than once. However, any ephemeral units in the group are loaded and executed again. If an ephemeral unit should run only once, have the unit set a scripted global variable or instance variable that can be tested by your initialization code.

Processing Individual Units

Instead of processing a loadable group all at once, you can selectively load individual units. In that case, you must get the LoadableGroup and LoadableUnit objects from the loader that represent the units. The code below provides an example of this technique.

```
myLoader := new Loader
myGroup := getGroup myLoader "ldgroup"
unit1 := getLoadableUnit myGroup "firstUnit"
unit2 := getLoadableUnit myGroup "secondUnit"
firstID := loadModule myLoader myGroup unit1
secondID := loadModule myLoader myGroup unit2
```

In this example, two units, firstUnit and secondUnit, have been loaded from the ldgroup loadable group. The Loader class defines the loadModule method, which brings these units into memory and invokes their entry point routines. (If loadModule cannot return the specified unit, it returns a global instance of LoadableUnitID that represents an error condition.) Note that individual units are loaded irrespective of the setting of their load list flag.

Once a unit has been loaded into memory, you can examine the value returned by its entry point function by calling the loaderValue method on the Loader instance and passing the unit ID as a parameter. The following example assumes that the entry point function of unit firstID returns a value that indicates whether or not initialization was successful.

```
success := (loaderValue myLoader firstID)
```

When loaderValue is called on an ephemeral unit, it always returns empty. To make an ephemeral unit's entry point function return a value, call loaderValue again, before any other loadable unit has run, passing the value empty as the unit ID. In the following example, the second call to loaderValue is able to inspect the return value that is left over from the first call, provided that one other loadable units have run through the same loader.

```
-- the first call to loaderValue returns empty,
success := loaderValue myLoader firstID
-- the second call returns a value
success := loaderValue myLoader empty
```

An ephemeral unit has other mechanisms for communication with ScriptX. It may set a global variable, set properties, change the state of an instance of `MemoryObject`, or throw an exception to indicate its completion status.

Releasing Loadable Units

When a unit is no longer needed in memory, call the `relinquish` method on the `LoadableUnitID` object. (The `LoadableUnitID` object is then garbage collected like any other ScriptX object.) The loadable unit is not physically removed from memory until all references to it are discarded. The following lines show examples of using the `relinquish` method to release a loadable unit.

```
forEach i in idList do relinquish i
relinquish firstID
```

If you have defined an exit point function in your loadable extension, that function will be called as part of the relinquish process.

Writing Loadable Extensions

While most applications and tools can be written directly in the ScriptX language, it is sometimes necessary to program at a lower level than allowed by the scripter. This can be done by using the C programming language. Using a loadable extension written in C, you have access to hardware features such as I/O ports and devices. With extensions, you can create device controllers, importers and exporters, or high-speed computational functions.

The code you write to create extensions can interact with the ScriptX object system, and has access to any capability provided at the scripter level.

Loadable extensions must be compiled separately on each platform. The ScriptX debugger cannot be used with loadable extensions—a system-level debugging tool is required. For information on compiler options and debugging tools for each ScriptX platform, see Appendix A, "Platform Notes."

Combining ScriptX and External Code.

The ScriptX run-time environment supports loading code that is not part of the system by using an instance of the `Loader` class and associated classes, as described in the first part of this chapter.

The loader resolves external references in the loaded code to symbols in the ScriptX substrate. You can, therefore, invoke ScriptX methods and refer to ScriptX objects inside loadable extensions. The interface between loadable extensions and the ScriptX substrate is described in "API for ScriptX Loadable Extensions" on page 333" later in this chapter.

Creating a Loadable Unit

Version 1.0 of the ScriptX Language and Class Library restricted loadable units to processing data passed in from the substrate. With the version 1.0 API, a loadable unit could not modify the display or make operating system calls that would conflict with the operation of the ScriptX environment.

Version 1.5 of the ScriptX Language and Class Library extends the API for loadable extensions, allowing a program to access any capability provided at the scripting level without having to "pop up" to that level to perform the needed function. With the version 1.5 API, access is provided via an external call function that boxes C arguments into a parameter block and passes it in to the substrate using the same mechanisms provided for scripted code. Thus, all arguments and return values used with loadable extensions are objects.

Using the version 1.5 API, extensions can create objects using the services of ScriptX, but they cannot store these object locally. A name-based interface is provided for creating and accessing ScriptX objects. Function calls refer to ScriptX global objects using module name and object name, passing name arguments as C strings.

For a complete listing of the API, see "API for ScriptX Loadable Extensions" on page 333. The following rules apply to programming loadable extensions in C:

1. Both functions and data can be declared in C. However, loadable extensions have access to objects only through the services of ScriptX. Static data cannot be of type `SXObject`, nor can it be a structure with elements of type `SXObject`.
2. Functions declared in C can be exported to the scripter as global functions (primitives) or as generic functions (methods), using the functions `SXextMakeGlobal` and `SXextMakeGeneric`. Such functions may accept from 0 to 10 arguments, all of which must be of type `SXObject`.
3. Generic functions created in C cannot override generics that are created in the ScriptX substrate. (Generics created in the ScriptX substrate are visible in the substrate module, whose interface module is the ScriptX module.) However, generic functions created in C are allowed to overload generics that are created in the scripter.
4. A ScriptX object (any value of type `SXObject`) cannot be stored into any statically declared C variable. To store a ScriptX object, store it as a scripter level global variable or instance variable, using the functions provided for that purpose (`SXextGetGlobal`, `SXextSetGlobal`, `SXextGetIV`, and `SXextSetIV`).
5. You may declare stack (automatic) variables of type `SXObject` and store objects in them.
6. You may use the standard C functions (`malloc` and `free`) to allocate and free dynamic memory that is not shared at the scripter level.
7. To make C structures that are visible at the scripter level, you must create instances of class `MemoryObject`. A `MemoryObject` object can be created either from the scripter or from a loadable extension, but its methods can be called only from C. The complete API for the `MemoryObject` class is given on page 341. Because they are objects, they must be stored according to the rules defined here.
8. You can create a `MemoryObject` either by specifying a size in bytes, or by specifying the structure components and names in a special array. Structure components must be of one of the following types: `SXchar`, `SXshort`, `SXint`, `SXdouble`, and `SXstring`. These types are defined by OIC and exported to the external API in the header file `SXExtend.h`.
9. A loadable extension in C can directly invoke any function in the `SXextend.h` file. It can invoke any function or method defined in the*ScriptX Class Reference* via the `SXextCall` function.
10. ScriptX runs essential system services such as garbage collection, callbacks, and event dispatch in other threads, which run concurrently with the thread from which the entrypoint function was called. If a loadable extension runs for a long time, it should call `SXthreadYield` or `SXthreadIdle` at reasonable intervals to permit ScriptX to run.

Examples of Loadable Extensions

The following examples illustrate how to create loadable extensions in C. For the complete extending ScriptX API see “API for ScriptX Loadable Extensions” on page 333.

A Simple Example

This simple example, which requires only the original ScriptX Version 1.0 API, is the “Hello World!” of loadable units.

```
#include "SXextend.h"
SXobject startHere(SXobject id, SXobject grp, SXobject unit)
{
   return SXwriteString (debug, "Hello World!\n");
}
```

The code in the previous example defines a single entry point named `startHere`. When this function is called, the OIC function `SXwriteString` is invoked on the `debug` object with a sign-on string. `SXwriteString`, a generic function that is implemented by streams, is one of the functions that is exported from the substrate as part of the API for loadable extensions. Printing to the `debug` stream causes a string to appear in the console window of the authoring environment.

Functions mentioned in this chapter that have names beginning with `SX`, such as `SXWriteString`, are defined in OIC in the ScriptX substrate, and exported for use in loadable extensions. Prototype definitions for the available OIC functions are declared in the header file `SXextend.h`.

Because this code has no further use once it has signed on, it is a likely candidate to be an ephemeral unit. (To make a unit ephemeral, set the sixth line in its registration in the group file to `LoaderTypeEphemeral`.) Another likely task for an ephemeral unit would be to check for the existence of a particular piece of hardware on the system and return `SXtrue` or `SXfalse` as a result.

Entry Points and Exit Points

Every loadable unit must have an entry point and may optionally have an exit point. ScriptX invokes the entry point function when the unit is loaded. If an exit point function exists, ScriptX invokes the exit point function when `relinquish` is called on the loadable unit. The example above has only an entry point function.

The entry and exit functions you create must conform to the prototypes shown below:

```
SXobject entryPt(SXobject ld, SXobject grp, SXobject unit);
void exitPt(SXobject env, char *gName, char *uName);
```

An entry point function is called with three parameters, the loader, group, and unit objects that are active in loading your code. Entry point functions written in C have no way of using these objects. Do not attempt to modify them in any way. An entry point function should return a legal ScriptX object.

An exit point function is called with three parameters also. The first is the *environment*, which is the value your entry point function returned. The second and third arguments are C strings that identify the names of the loadable group and loadable unit in which your code is contained. Your exit point function (which is completely optional) should not return a value.

Exporting Functions to the Scripter

This example shows how to create a global function and export the function and a scripter name to a particular ScriptX module. Use this example as a prototype for creating an external routine that is coded in C for computational efficiency.

The ScriptX Primitive class defines the behavior of executable code objects. Global functions such as the comparison functions are implemented as instances of Primitive in ScriptX. A Primitive object holds the address of an executable function and the minimum and maximum number of parameters the function requires. The function SXextMakeFunction creates a Primitive object and binds it to a name. Primitives and other objects created by an extension can be bound to global variables in the scripter, in any particular module.

The following example shows an extension which defines a factorial function. In this case, the entry point code exports the function to the scripter and the factorial function remains in memory to be called when needed. For a complete listing of the header file SXextend.h, see page 343.

```
#include "SXextend.h"

/*
   Factorial function: to demo use of loader
   The following script loads and binds the code:
      ld := new loader
      grp := getgroup ld "<groupdirname>"
      unit := getloadableunit grp "<unitname>"
      myid := loadmodule ld grp unit
      fact := loadervalue ld myid
      fact 7
*/

/*
 * Factorial:  computes x!  Uses floating point to
   allow large results.
*/
SXdouble factorial(SXdouble d)
{
   if (d < 2.0)
      return 1.0;
   else
      return d * factorial(d - 1.0);
}

/*
 * This is the function that we export to the scripter.  It
 * converts between the object world and the C data world.
*/


SXobject factFn(SXobject n)
{
   return SXdoubleToObject(factorial(SXdoubleFrom(n)));
}
```

```
/*
 * Loader entry:  sign on, export our symbol.
 */
SXobject factEntryPoint(SXobject id, SXobject grp, SXobject unit)
{
SXobject s;

    s = SXmakeString("Hello from Factorial!\n");
    SXwriteString(debug, "Hello from Factorial!\n");;
    return SXextMakeFunction("numericsImplementation", "factorial",
        factFn, 1, 1);
}
```

In this example, the `factEntryPoint` exports the function as a code primitive. To export an extension function to the scripter, you must create a `Primitive` object by using the `SXextMakeFunction` function and pass that object back to ScriptX as the result of your entry point function.

From the scripter, the entry point function's return value is retrieved by calling `loaderValue` on the current `Loader` instance.

The following ScriptX code loads the unit and binds the exported function pointer to a scripter symbol.

```
myLoader := new loader
grp := getGroup myLoader "factgroup"
unit := getLoadableUnit grp "factorialUnit"
myid := loadModule myLoader grp unit
factorial := loaderValue myLoader myid
factorial 7
```

This example assumes that the loadable unit's group file is in a directory called `factgroup`, and that the unit name is `factorialUnit`. The call on `loaderValue` retrieves the value returned by the entry point function last loaded from the loader. A possible group file for this extension might look like this:

```
#define factorial extension
factorial
factorl.lib
factEntryPoint
0
1
LoaderTypeLinkable
true
```

When you no longer need your factorial function, The `relinquish` method informs the loader that you no longer need the loadable unit, and that it can be released from memory. To release the unit in the preceding example, you must call `relinquish` on the unit id, and destroy the pointer to the primitive function object.

```
factorial := undefined
relinquish myID
```

When you export values that point into loaded code to the scripter (such as the factorial function exported in the previous example), be sure to remove any references before you relinquish the loadable extension. For example, if you executed relinquish myID, and then called factorial again, the function pointer in the factorial primitive would no longer be valid, and the system would probably crash.

Access to ScriptX from Loadable Units

Version 1.5 of ScriptX adds additional functionality to the Extending ScriptX API, including the MemoryObject class. An instance of MemoryObject is an object, a fixed chunk of memory, that both ScriptX and C code can have a pointer to. This opens the way for additional communication between loadable extensions and ScriptX.

The following example creates a subclass of MemoryObject and defines methods in C for logical operations on the new class. It is typical of loadable units that define certain operations in C, for computational efficiency. Note that a BitArray object can implement generic functions that are coded in ScriptX and generic functions that are coded in C. In this example, the loadable unit has access to ScriptX from within C.

The first portion of this example is written in ScriptX, and it creates a memory object, an instance of a subclass of MemoryObject called BitArray. Initialization of the BitArray class is in ScriptX. Other generics for operating on a bit array can be defined in C and imported into ScriptX as generic functions using the helper function SXextMakeGeneric.

```
--  ScriptX part of BitArray example
class BitArray (MemoryObject)
    inst vars
        bitSize
    class methods
        method afterInit self #rest args -> (
            apply nextMethod self args
            process (new loader) "exsx2"
        )
end

method init self {class BitArray} #rest args #key numBits:(16) -> (
    if (mod numBits 8 !== 0) do
        report generalError #("numBits must be a multiple of 8\n")
    apply nextMethod self initialSize:((numBits / 8) as Integer) args
    self.bitSize := numBits
    clear self
)
-- specialize localEqual to provide for equality test
method localEqual self {class BitArray} other -> (
    bitequal self other
)
-- specialize prin to provide for printing
method prin self {class BitArray} how strm -> (
    for i := 0 to (self.bitSize - 1) do
    writeString strm ((getbit self i) as String)
)
```

The remaining code is compiled in C, and loaded into ScriptX at run time. This simple example defines an entry point function, a utility function that locks the memory object and returns a pointer, and a series of standard operations for manipulating bits. Each function has a prototype as a ScriptX object (type SXobject).

The entrypoint function includes a series of calls to SXextMakeGeneric, one of the Extending ScriptX API helper functions. Each call to SXextMakeGeneric creates a generic function, and a method that implements that generic function on the BitArray class. For example, the "and" bit operation is defined as the C function andOp and exported to the scripter as the generic function logicalAnd, implemented as a method on BitArray. For a defintion of SXextMakeGeneric, see page 338.

```
#include "SXextend.h"
#include <memory.h>
#include <string.h>

/* C support for BitArray class - simplified version */
#define    SYS_MODULE    NULL
#define    BIT_MODULE    "scratch"

typedef enum {
    OPequal,
    OPand,
    OPor,
    OPxor
} binaryOpcode ;

/* prototypes */
void *getPointer(SXobject arg, SXint *objSize);
SXobject binaryOp(SXobject self, SXobject arg2, binaryOpcode op);
SXobject equalOp(SXobject self, SXobject arg2);
SXobject andOp(SXobject self, SXobject arg2);
SXobject orOp(SXobject self, SXobject arg2);
SXobject xorOp(SXobject self, SXobject arg2);
SXobject notOp(SXobject self);
SXobject clearOp(SXobject self);
SXobject setBitOp(SXobject self, SXobject which, SXobject val);
SXobject getBitOp(SXobject self, SXobject which);
SXobject entryPoint(SXobject ld, SXobject grp, SXobject uni);

/* lock down the external memory object and return it's pointer */
void *getPointer(SXobject arg, SXint *objSize)
{
    *objSize = SXintFrom(SXextGetIV(BIT_MODULE, arg, "bitSize"));
    SXlockMem(arg);
    return SXdereference(arg);
}

/* compare two BitArrays */
SXobject binaryOp(SXobject self, SXobject arg2, binaryOpcode op)
{
    register unsigned char*objp, *otherp;
    SXint             objSize, otherSize;
    SXint             opResult;
    SXobject      arg1, result;

    if (op == OPequal) {
        result = trueObject;
        arg1 = self;
    }
    else {
        /* copy of self is destination */
        result = SXextCall(SYS_MODULE, "copy", self, NULL);
        arg1 = result;
    }
    objp = getPointer(arg1, &objSize);
    otherp = getPointer(arg2, &otherSize);
    if (objSize != otherSize) {
        SXunlockMem(arg1);
        SXunlockMem(arg2);
        SXextCall(SYS_MODULE, "report", generalError,
            SXmakeString("BitArray objects must have same size"), NULL);
        return SXundefined;
    }

    objSize /= 8;
    while (objSize--) {
        switch (op) {
            case OPequal :
                if (*objp != *otherp) {
                    result = falseObject;
                    goto bailOut;
                }
                break;
            case OPand :
                *objp &= *otherp;
                break;
            case OPor :
                *objp |= *otherp;
                break;
            case OPxor :
                *objp ^= *otherp;
                break;
        } /* switch */
        objp++;
        otherp++;
    } /* while*/
    /* bailOut is a label, target of a goto from within the loop */
    bailOut:
    SXunlockMem(arg1);
    SXunlockMem(arg2);
    return result;
}

/* compare two BitArrays */
SXobject equalOp(SXobject self, SXobject arg2)
{
    return binaryOp(self, arg2, OPequal);
}

/* and two BitArrays */
SXobject andOp(SXobject self, SXobject arg2)
{
    return binaryOp(self, arg2, OPand);
}

/* or two BitArrays */
SXobject orOp(SXobject self, SXobject arg2)
{
    return binaryOp(self, arg2, OPor);
}

/* xor two BitArrays */
SXobject xorOp(SXobject self, SXobject arg2)
{
    return binaryOp(self, arg2, OPxor);
}

/* logical not of a BitArrays */
```

```
SXobject notOp(SXobject self)
{
    unsigned char*objp;
    SXint      objSize;
    SXobject result;

    /* copy of self is destination */
    result = SXextCall(SYS_MODULE, "copy", self, NULL);

    /* get pointer to destination */
    objp = getPointer(result, &objSize);

    objSize /= 8;
    while (objSize--) {
        *objp++ = ~ *objp;
        }

    SXunlockMem(result);
    return result;
}

/* zero out a BitArray */
SXobject clearOp(SXobject self)
{
    unsigned char*objp;
    SXint      objSize;

    /* get pointer to destination */
    objp = getPointer(self, &objSize);

    objSize /= 8;
    while (objSize--) {
        *objp++ = 0;
    }

    SXunlockMem(self);
    return self;
}

/* write nth bit, zero-based, bit 0 is leftmost */
SXobject setBitOp(SXobject self, SXobject which, SXobject val)
{
    unsigned char*objp;
    SXint      objSize;
    SXint      bit;
    SXint      result;

    /* get pointer to self */
    objp = getPointer(self, &objSize);
    bit = SXintFrom(which);
    if (bit < 0 || bit >= objSize) {
        SXunlockMem(self);
        SXextCall(SYS_MODULE, "report", generalError,
            SXmakeString("BitArray index out of bounds"), NULL);
        return SXundefined;
    }
    if (val == SXintToObject(0) || val == falseObject)
        objp[bit / 8] &= ~ (1 << (7 - (bit % 8)));
    else
        objp[bit / 8] |= (1 << (7 - (bit % 8)));

    SXunlockMem(self);
    return self;
}

/* read nth bit, zero-based, bit 0 is leftmost */
SXobject getBitOp(SXobject self, SXobject which)
{
    unsigned char*objp;
    SXint      objSize
    SXint      bit;
    SXint      result;

    /* get pointer to self */
    objp = getPointer(self, &objSize);
    bit = SXintFrom(which);
    if (bit < 0 || bit >= objSize) {
        SXunlockMem(self);
        SXextCall(SYS_MODULE, "report", generalError,
            SXmakeString("BitArray index out of bounds"), NULL);
        return SXundefined;
    }

    if (objp[bit / 8] & (1 << (7 - (bit % 8))))
        result = 1;
    else
        result = 0;

    SXunlockMem(self);
    return SXintToObject(result);
}

/* function executed when this library is loaded */
SXobject entryPoint(SXobject ld, SXobject grp, SXobject uni)
{
    SXobjectmyClass;
    myClass = SXextGetGlobal(BIT_MODULE, "BitArray");
    if (myClass == SXempty) {
        SXextCall(SYS_MODULE, "report", generalError,
            SXmakeString("Class BitArray not found"), NULL);
        return falseObject;
    }
    /* specialize BitArray with new methods */
    SXextMakeGeneric(BIT_MODULE, "bitEqual", myClass, equalOp);
    SXextMakeGeneric(BIT_MODULE, "bitAnd", myClass, andOp);
    SXextMakeGeneric(BIT_MODULE, "bitOr", myClass, orOp);
    SXextMakeGeneric(BIT_MODULE, "bitXor", myClass, xorOp);
    SXextMakeGeneric(BIT_MODULE, "bitNot", myClass, notOp);
    SXextMakeGeneric(BIT_MODULE, "clear", myClass, clearOp);
    SXextMakeGeneric(BIT_MODULE, "getBit", myClass, getBitOp);
    SXextMakeGeneric(BIT_MODULE, "setBit", myClass, setBitOp);

    return trueObject;
}
```

Access to C Data Structures from ScriptX

A loadable unit can also define its own data structures that are not ScriptX objects. (The main limitation on loadable units is that they cannot have static access to ScriptX objects.) A useful application of this example would be to create a buffer variable in an importer.

The Example class creates a loader and calls a loadable unit from ScriptX in its afterInit method.

```
-- ScriptX portion of class
class Example()
   class vars
      loadID
   inst Vars
      x, y
      cstruct
   class methods
      method afterInit self #rest args -> (
         apply nextMethod self args
         process (new Loader) "example"
   )
end

method afterInit self {class Example} #rest args -> (
   /* create instance of MemoryObject */
   self.cstruct := allocateExample()
   apply nextMethod self args
)
/*  accesses internal C structures in a memory object */
method doStuff self {class Example} -> (
   self.cstruct[@accessCode] := 5
   self.cstruct[@accessMethod] := "FTP"
   doConnect self.cstruct
)
```

The C portion of the Example class definition is compiled to example.lib.

```
#include SXextend.h
typeDef mystruct struct {
   SXint         accessCode;
   SXchar        accessMethod[256];
   int           other;
   int           stuff;
} mystruct;

SXobject allocFn(SXobject mem)
{
   /*     EXT_INT AND EXT_STRING
          are defined in the header file SXextend.h */
   return SXmakeMemoryObject(EXTsize(mystruct),
      EXTtype("accessCode", EXT_INT, mystruct, accessCode),
      EXTtype("accessMethod", EXT_STRING, mystruct, accessMethod),
      null);
}

SXobject connectFn(SXobject ld, SXobject grp, SXobject uni)
{
   mystruct *mp;
```

```
    SXlockmem(mem);
    mp = SXdereference(mem);
    doStuff(mp);
    SXunlockmem(mem);
    return trueObject;
}

/*  the entrypoint function exports C functions to the scripter
*   as primitives (global functions) */
SXobject entryPoint(SXobject ld, SXobject grp, SXobject uni)
{
    SXextMakeFunction("xmplImpl", "doConnect", connectFn, 1, 1);
    SXextMakeFunction("xmplImpl", "allocateExample", allocFn, 0, 0);
}
```

Common Errors

The following is a list of the most common mistakes in using the ScriptX external API:

1. The most common error in using the ScriptX external API is the specification of the wrong module in a helper function. Every ScriptX name is a symbol that is defined in a particular module. A module can determine whether names are visible to other modules, and can export all names it defines, or selected names. If the variable that defines a name exports it, other modules must decide explicitly whether or not they import names from any given module, and which name they import. As an optimization, ScriptX interprets a null value as the ScriptX module, which is the interface module for the ScriptX substrate, in which the core classes are defined.
2. Only object values can be passed to and from ScriptX. Passing a non-object value causes ScriptX to report an exception.
3. The helper function SXextCall, used to call a ScriptX function or method from C, requires a statement terminator. A common error is the failure to terminate a list of arguments to SXextCall.

API for ScriptX Loadable Extensions

The ScriptX external API is defined in three parts. First, there is a set of function and type definitions from the substrate that are exported to the external API. Second, there is a set of helper functions that map C calls into substrate calls. Finally, the external API defines a new class (MemoryObject) for representing allocated memory blocks.

This section provides a reference to the types and functions exported to the external API in the header file SXextend.h. Each entry identifies the version number of ScriptX for which it was first added to the external ScriptX API. Use this information, when needed, to write code that is compatible with earlier versions of the Kaleida Media Player.

Functions that are defined in the substrate and exported to the ScriptX external API are prefixed by SX. Functions that are defined as part of the ScriptX external API are prefixed by SXext.

Type Definitions

Extensions can use these data types to communicate with the ScriptX substrate.

SXint (1.0)

```
typedef long SXint;
```

The `SXint` type is a 32-bit single integer value.

SXbool (1.5)

```
typedef SXint SXbool;
```

The `SXbool` type is a 32-bit single integer value, where a value of zero represents `false` and a non-zero value represents `true`.

SXdouble (1.0)

```
typedef double SXdouble;
```

The `SXdouble` type is a floating point value that implements the 64-bit IEEE floating point standard. For more information on fixed and floating point arithmetic in ScriptX, see the "Numerics" chapter of the *ScriptX Components Guide*.

SXobject (1.0)

```
typedef void* SXobject;
```

The `SXobject` type represents any ScriptX object. All objects are passed by pointer, so `void*` is used as the declaration for object values.

Global Constants and Variables

debug (1.0)

The global variable *debug* is a `Stream` object. When you write to the `debug` stream in the ScriptX authoring environment, the output appears on the debug console.

SXempty (1.5)

A ScriptX system object that represents the state of emptiness on a collection or container. Never set the value of any ScriptX object to `SXempty`; it is a value that is returned by the system.

falseObject (1.0)

The ScriptX `Boolean` object representing the value *false*.

SXundefined (1.0)

The ScriptX object representing the value *undefined*.

SXunsupplied (1.5)

The ScriptX object representing the value *unsupplied*.

trueObject (1.0)

The ScriptX `Boolean` object representing the value *true*.

Entry/Exit Prototypes

entryPoint (1.0)

`SXobject` *entryPoint* `(SXobject` *loader,* `SXobject` *group,* `SXobject` *unit*`)`

entryPoint	Name of the entry function
loader	The `Loader` object
group	The `LoadableGroup` object
unit	The `LoadableUnit` object

Your function must return a legal ScriptX object. For more information on defining an entry point function, see the discussion of “Entry Points and Exit Points” on page 324. For a code example that demonstrates how to define a legal entry point function, see `SXextMakeFunction` on page 338.

exitPoint (1.0)

`SXobject` *exitPoint* `(SXobject` *env,* `char *`*gName,* `char *`*uName*`)`

exitPoint	Name of the exit function
env	Return value of the entry function, an object
gName	The name of the loadable group
uName	The name of the loadable unit

Performs any necessary operations before the loadable unit is released. For more information on defining an exit point function, see the discussion of “Entry Points and Exit Points” on page 324.

Conversion Functions

SXintToObject (1.0)

`SXobject SXintToObject(SXint` *val*`)`

val	The 32-bit integer to be converted

Returns a ScriptX object representing the value. For an example in which `SXintToObject` is used, see `SXmakeGlobal` on page 339.

SXdoubleToObject (1.0)

`SXobject SXdoubleToObject(SXdouble` *val*`)`

val	The 64-bit floating point value to be converted

Returns a ScriptX object representing the value.

SXmakeString (1.0)

`SXobject SXmakeString(char *`*str*`)`

str	A C string

Returns a ScriptX object representing the string. The following example demonstrates the use of `SXmakeString` to report an exception in ScriptX:

```
SXObject x;
x = SXextCall(SYS_MODULE, "report", generalError,
SXmakeString("Bad craziness in this code!"), NULL);
```

SXintFrom (1.0)

`SXint SXintFrom(SXobject` *val*`)`

val	A ScriptX `Number` object.

Returns a 32-bit integer with the same integer value as the ScriptX object. The usual rules about truncation apply. For more information, see the "Numerics" chapter of the *ScriptX Components Guide*.

SXdoubleFrom (1.0)

`SXdouble SXdoubleFrom(SXobject` *val*`)`

val	The ScriptX object to be converted

Returns a C floating point value equivalent to *val*. For more information, see the "Numerics" chapter of the *ScriptX Components Guide*.

SXstringOf (1.0)

`char * SXstringOf(SXobject` *str*`)`

str	The ScriptX string object.

Returns a pointer to the C string represented by the given ScriptX object *str*. Do not modify the string in any way. Make a copy of the string if you need to modify it. Any ScriptX `String`, `StringConstant`, or `Text` object can be used as an argument to `SXstringOf`.

Thread Functions

SXthreadIdle (1.0)

`void SXthreadIdle()`

Allows any necessary system requests to be served. Other threads may be scheduled during the execution of this call.

SXthreadYield (1.0)

`void SXthreadYield()`

Allows ScriptX to switch to the next scheduled thread.

Helper Functions

SXextCall (1.5)

```
SXobject SXextCall(char *module, char *functionName ,
         SXobject object1, SXobject object2, ..., NULL)
```

module	The module in which the call is made
functionName	The name of the generic or global function to call
object1, object2, . . .	A variable length list of from 0 to 10 objects, passed as arguments to the generic or global function *functionName*.
`NULL`	`NULL` is required to terminate the argument list

Calls a given global or generic function in the given module, passing the list of objects as arguments.

The following code example demonstrates how to use `SXextCall` to call a generic function in ScriptX:

```
result = SXextCall(SYS_MODULE,"addEventInterest",
SXextGetIV("scratch",self,"mouseMoveInterest"),NULL);
```

The syntax for calling a global function is the same. For an example that demonstrates how to report an exception, see `SXmakeString` on page 335. For a code example that demonstrates how to call a function or method with keyword arguments, see `SXextMakeGlobal` on page 339.

SXextGetGlobal (1.5)

`SXobject SXextGetGlobal(char` **moduleName,* `char` **globalName*`)`

moduleName	The module in which the call is made
globalName	Name of the given global variable

Returns the value of the given global variable *globalName* that is defined in the module *moduleName*.

The following example demonstrates how to access a ScriptX global variable. First, the global variable `myMouseEvent` is defined in the scripter.

```
-- ScriptX portion of example
in module special_module
global flapPresenter := new TwoDShape
object myMouseEvent (MouseUpEvent) presenter:flapPresenter end
```

The global variable `myMouseEvent` can be accessed from a loadable extension using `SXextGetGlobal`.

```
/* C portion of example */
SXobject mouseevent_1;
mouseevent_1 = SXextGetGlobal("special_module", "myMouseEvent");
```

Note that names of classes are global names in the ScriptX module, and can be referenced using `SXextGetGlobal`. See the code example given for `SXmakeNameInterned` on page 339.

SXextGetIV (1.5)

`SXobject SXextGetIV(char` **moduleName,* `SXobject` *obj,* `char` **iv*`)`

moduleName	The module in which the call is made
obj	A ScriptX object
iv	Name of the given instance variable on *obj*

Calls a given global or generic function in the given module, passing the list of objects as arguments.

The following example demonstrates how to access a ScriptX instance variable from C. This instance variable is defined by the object `mouseevent_1`, a ScriptX `MouseUpEvent` object, in the example given for `SXextGetGlobal`.

```
/* this example continues the example for SXextGetGlobal */
SXobject presenter_1;
presenter_1 = SXextGetIV("special_module", mouseevent_1, "presenter");
```

SXextGetModule (1.5)

```
SXobject SXextGetModule(char *moduleName)
```

moduleName	The name of a module

Returns a `ModuleClass` object with the name *moduleName*, or `falseObject` if no such module exists. `SXextGetModule` is equivalent to the global function `getModule` in the scripter, which is defined in the “Global Functions” chapter of the *ScriptX Class Reference*.

SXextMakeFunction (1.5)

```
SXobject SXextMakeFunction(char *module, char *fnName, SXobject (*func)(),
          SXInt minArgs, SXInt maxArgs)
```

module	The module in which the function is defined
fnName	The name of the function in that module
func	A pointer to a C function
minArgs	Integer representing minuimum number of arguments
maxArgs	Integer representing maximum number of arguments

Creates a ScriptX `Primitive` object that represents the given C function *func* and binds that function to a ScriptX name. The funtction is callable from the scripter. The `Primitive` class, a subclass of `AbstractFunction`, represents a ScriptX global function.

The following example demonstrates the use of `SXextMakeFunction`. The function `bowdlerize` is first defined in C; it takes one argument, which must be of type `SXobject`. The script then calls `SXextMakeFunction` to export `bowdlerize` to the scripter, assigning it as the entry point function. From the scripter, `bowdlerize` is called as a global function.

```
/* first define the function in C */
SXobject bowdlerize(SXobject theText)
{
/* body of function, function processes text here */
}
/* now use SXextMakeFunction to export bowdlerize to the scripter */
SXobject entryPoint(SXobject ld, SXobject grp, SXobject uni)
{
    SXextMakeFunction("CensorImplementation", "bowdlerize",
        bowdlerize, 1, 1);
    return trueObject;
}
```

SXextMakeGeneric (1.5)

```
SXobject SXextMakeGeneric(char *module, char *generic,
          SXobject specializer, SXobject (*func ())
```

module	The module in which the generic is defined
generic	The name of the generic in that module
specializer	The object that is specialized by the generic
func	A pointer to a function

Creates a ScriptX generic function, defined in the given *module*, with the name *generic* that is defined in that module. The function is implemented as a method defined by *specializer*, which is a ScriptX class or object. The function *func*, which implements a method for the generic function, will create an instance of one of the subclasses of `AbstractFunction` in ScriptX. (The actual function class is not specified, and is subject to change in future versions of ScriptX.)

The following example demonstrates the use of SXextMakeGeneric. The generic function bowdlerize is first defined in C; it takes one argument, which must be an object of type SpecialText. The script then calls SXextMakeGeneric to export bowdlerize to the scripter, assigning it as the entry point function. From the scripter, bowdlerize is called as a method on SpecialText.

```
/* first define the function in C */
SXobject bowdlerize(SXobject self)
{
/* body of method, which processes a SpecialText object */
}
/* now use SXextMakeGeneric to export bowdlerize to the scripter */
SXobject entryPoint(SXobject ld, SXobject grp, SXobject uni)
{
SXobject   myClass;
myClass = SXextGetGlobal("scratch", "SpecialText");
    if (myClass == SXempty) {
    SXextCall(SYS_MODULE, "report", generalError,
        SXmakeString("Class SpecialText not found"), NULL);
    return falseObject;
}
/* export "bowdlerize" to the scripter as a generic function */
SXextMakeGeneric("scratch", "bowdlerize", myClass, bowdlerize);
return trueObject;
}
```

SXextMakeGlobal (1.5)

SXobject SXextMakeGlobal(char **module,* char **global,* SXobject *value*)

module	The module in which the global is defined
global	The name of the global in that module
value	The object that is assigned to the global

Defines a ScriptX global variable with the given name, in the given module, and assigns its value.

The following example uses the C interface to create a new instance of the ScriptX class RGBColor, and assigns it to the global variables bgColor.

```
bgColor = SXextCall(SYS_MODULE,"new",
SXextGetGlobal(SYS_MODULE,"RGBColor"),
SXmakeNameInterned("red"), SXintToObject(0),
SXmakeNameInterned("green"), SXintToObject(100),
SXmakeNameInterned("blue"), SXintToObject(100),NULL);
SXextMakeGlobal(SYS_MODULE,"bgColor",bgColor);
```

SXmakeNameInterned (1.5)

SXobject SXmakeNameInterned(char **str*)

str	A pointer to a C string

Creates a ScriptX NameClass object, given a C string *str,* and interns the name in the system name table.

The following example demonstrates the use of SXmakeNameInterned in calling a ScriptX generic function with keyword arguments:

```
SXObject myRect;
myRect = SXextCall(SYS_MODULE,"new",
SXextGetGlobal(SYS_MODULE,"Rect"),
SXmakeNameInterned("x2"),SXintToObject(320),
SXmakeNameInterned("y2"),SXintToObject(240),NULL);
```

SXreadStream (1.5)

`SXobject SXreadStream(SXobject` **stream,* `void*`*buffer*
        `SXint` *length*`)`

stream	`Stream` object, a ScriptX stream
buffer	A C buffer, into which the stream is read
length	An integer, the length of the stream in bytes

Reads safely from the given *stream* of the given *length* in the given *buffer.*

SXextSetGlobal (1.5)

`SXobject SXextSetGlobal(char` **moduleName,* `char` **globalName*
        `SXobject` *value*`)`

moduleName	The module in which the call is made
globalName	Name of the given global variable
value	The value to be set, a ScriptX object

Sets the value of the global variable *globalName,* in module *moduleName,* to the given *value,* which must be of type `SXobject`.

SXextSetIV (1.5)

`SXobject SXextSetIV(char` **moduleName,* `SXobject` *obj,* `char` **iv*
        `SXobject` *value*`)`

moduleName	The module in which the call is made
obj	A ScriptX object
iv	Name of the given instance variable on *obj*
value	The value to be set, a ScriptX object

Sets the value of the instance variable *iv,* defined by object *obj,* in module *moduleName,* to the given *value,* which must be of type `SXobject`.

The following example sets the value of the instance variable `pattern`, defined by the object `fBrush`, to `grayPattern`, which is a global instance of `Brush`.

```
SXextSetIV(SYS_MODULE,fBrush,"pattern",
SXextGetGlobal(SYS_MODULE,"grayPattern"));
```

SXwriteStream (1.5)

`SXobject SXwriteStream(SXobject` **stream,* `void*`*buffer*
        `SXint` *length*`)`

stream	`Stream` object, a ScriptX stream
buffer	A C buffer, into which the stream is read
length	An integer, the length of the stream in bytes

Writes safely from the given *buffer* into the given *stream,*

SXwriteString (1.0)

`void SXwriteString(SXobject` *stream*`, void *` *cstring*`)`

stream	`Stream` object, for example, the `debug` object
cstring	A C string

Writes the string to the designated stream.

Creating a Memory Object

An instance of MemoryObject can be created either from the scripter or from a loadable extension. MemoryObject defines an init method with one required keyword argument, initialSize. For an example in which an instance is created from the scripter, see the BitArray class on page 327.

The function SXmakeMemoryObject creates an instance of MemoryObject from a loadable extension.

SXmakeMemoryObject (first calling sequence) (1.5)

```
void SXmakeMemoryObject(SXint size, NULL)
```

size	Integer object, the size in bytes
null	An ASCII null character, used as a terminator

Creates an instance of MemoryObject, a ScriptX object, of the given *size*. The null character acts as a terminator.

SXmakeMemoryObject (second calling sequence) (1.5)

```
void SXmakeMemoryObject(EXTsize(structure),
          [EXTtype(string, macro, structure, member),
          . . . ]

          NULL)
```

(EXTsize(*structure*)	A macro that determines the size in bytes
[EXTtype(), . . .]	A macro
null	Argument list terminator

Creates an instance of MemoryObject, a ScriptX object. The macros used in this calling sequence are defined in SXextend.h. The EXTsize macro, defined in SXextend.h, determines the size of the object, based on the size of the associated structure that is defined in C.

The EXTtype macro, also defined in SXextend.h, is an optional argument, and is invoked once for each structure member (field) that is imported from a structure defined in C to the scripter. It takes four arguments.

The first argument is a C string that will be used in the scripter as a name for that member. The second argument takes a macro for the underlying C data type, one of EXT_INT, EXT_CHAR, EXT_SHORT, EXT_LONG, or EXT_FLOAT. The third argument takes the name of the structure, the same name that is passed as an argument to EXTsize. The final argument is the C name of that structure member, as defined in the structure's typedef statement.

As in the first calling sequence to SXmakeMemoryObject, an ASCII null is required for the final argument. Since SXMakeMemoryObject accepts a variable number of invocations of EXTtype as arguments, this final argument acts as a terminator.

For an code sample that demonstrates the use of this calling sequence to SXmakeMemoryObject, see the Example class, which begins on page 332.

MemoryObject API

MemoryObject defines only a single method that is visible at the scripter level, the init method. MemoryObject defines the following instance methods, exported from the substrate to the ScriptX external API. Although the following generic functions are implemented in the substrate as methods on MemoryObject, they are called externally as functions in C. No scripter equivalents exist for these methods.

SXdereference (1.5)

```
void SXdereference(SXobject self)
```

self	`MemoryObject` object

Get the fixed address of the `MemoryObject` instance *self* in memory. A call to `SXdereference` must be preceded by a call to `SXlockMem`. If the memory object is locked, this call returns null.

SXgetOSHandle (1.5)

```
void SXgetOSHandle(SXobject self)
```

self	`MemoryObject` object

Get the handle representation of the `MemoryObject` instance *self* (system dependent). If the underlying memory system is handle-based, then a handle is returned. If the system is pointer-based, a pointer is returned.

SXlockMem (1.5)

```
void SXlockMem(SXobject self)
```

self	`MemoryObject` object

Locks the `MemoryObject` instance *self* in memory, so that the system cannot relocate it. A call to `SXlockMem` must precede a call to `SXdereference`.

SXreadAt (1.5)

```
void SXreadAt(SXobject self, SXint * offset
    void *buffer, SXint length)
```

self	`MemoryObject` object
offset	`Integer` object, position to begin reading
buffer	A C buffer
length	`Integer` object, number of bytes to read

Copies data safely from the `MemoryObject` instance self into a C buffer.

SXunlockMem (1.5)

```
void SXunlockMem(SXobject self)
```

self	`MemoryObject` objec

Unlocks the `MemoryObject` instance self in memory, allowing the system to relocate it. The number of calls to `SXunlockMem` must match and balance the number of calls to `SXlockMem` before the memory is relocatable.

SXwriteAt (1.5)

```
void SXwriteAt(SXobject self, SXint * offset
    void *buffer, SXint length)
```

self	`MemoryObject` object
offset	`Integer` object, position to begin reading
buffer	A C buffer
length	`Integer` object, number of bytes to read

Writes data safely from a C buffer into the `MemoryObject` instance *self*.

SXextend.h Header File

```
#include <stddef.h>

/*  Exported in V1.0 */

typedef long         SXint;
typedef SXint        SXbool;
typedef double       SXdouble;
typedef void         *SXobject;

extern SXobject      SXintToObject(SXint val);
extern SXobject      SXdoubleToObject(SXdouble val);
extern SXobject      SXmakeString(char *str);

extern SXint         SXintFrom(SXobject val);
extern SXdouble      SXdoubleFrom(SXobject val);
extern char*         SXstringOf(SXobject str);

extern SXobject      SXwriteString(SXobject stream, char *string);

extern void          SXthreadYield(void);
extern void          SXthreadIdle(void);

extern SXobject      trueObject;
extern SXobject      falseObject;

/**************************/

/*  Exported in V1.1 */
typedef SXobject     SXclass;
typedef char         SXchar;
typedef short        SXshort;

extern SXobject      generalError;
extern SXobject      SXempty;
extern SXobject      SXundefined;
extern SXobject      SXunsupplied;
extern SXobject      SXextDebugStream(void);

#define debug        (SXextDebugStream())

extern SXobject      SXmakeNameInterned(char *str);

/**************************/

/*  Helper functions defined for V1.1 */

extern SXobject      SXextMakeGlobal(char *moduleName,
                        char *globalName, SXobject val);
extern SXobject      SXextMakeGeneric(char *moduleName,
                        char *className, SXobject specializer,
                        SXobject (*func)());
extern SXobject      SXextMakeFunction(char *moduleName,
                        char *fnName, SXobject (*func)(),
                        SXint minargs, SXint maxargs);
extern SXobject      SXextGetModule(char *moduleName);
extern SXobject      SXextGetGlobal(char *moduleName,
                        char *globalName);
```

```
extern SXobject      SXextSetGlobal(char *moduleName, char *globalName,
                         SXobject val);
extern SXobject      SXextGetIV(char *moduleName,
                         SXobject obj, char *iv);
extern SXobject      SXextSetIV(char *moduleName,
                         SXobject obj, char *iv, SXobject val);
extern SXobject      SXextCall(char *moduleName,
                         char *generic, ... );

extern void SXextReadStream(SXobject strm, void *buffer, SXint length);
extern void SXextWriteStream(SXobject strm,
       void *buffer, SXint length);

/*  Accessor functions for MemoryObject class */

#define EXT_TYPE_SXCHAR 1
#define EXT_TYPE_SXSHORT 2
#define EXT_TYPE_SXINT 3
#define EXT_TYPE_SXDOUBLE 4
#define EXT_TYPE_SXN_CHAR 5
#define EXT_TYPE_SXCHAR_STAR 6

#define EXT_SIZE(baseType)(sizeof(baseType))
#define EXT_ELEMENT(sxname,sxtype,baseType,theField)\
       (sxname),((SXint)(sxtype)),((SXint)offsetof(baseType,theField))

extern SXobject      SXextMakeMemoryObject(SXint size, ... );
extern SXobject      SXreadAt(SXobject extmem,
                         void *buffer, SXint offset, SXint length);
extern SXobject      SXwriteAt(SXobject extmem,
                         void *buffer, SXint offset, SXint length);
extern SXint         SXlockMem(SXobject extmem);
extern SXint         SXunlockMem(SXobject extmem);
extern void          *SXdereference(SXobject extmem);
```

C-Language Development Environments

ScriptX supports one C-language compiler per platform for the development of C-extensions to ScriptX. On the Apple Macintosh, you must use the Symantec C++ compiler; on Microsoft Windows and OS/2, you must use the Watcom C/386 compiler. This chapter outlines the compile and link procedures required to create an extension on each platform.

Apple Macintosh 68XXX

See the printed documentation for current information.

Compiling an Extension

To create ScriptX extensions to use on the Macintosh, you must have Think C V7.0 or later.

Note – The Macintosh versions of ScriptX and the KMP are compiled with V7.0.1 of the Think C compiler.

For each extension, create a single source file and compile the module using the following switches:

```
Set Project Type
   ✓  Application
   ✓  Far Code
   ✓  Far Data
Compiler Settings
   ✓  Four-byte ints
   ✓  Align arrays of char
```

Linking the Extension

Use the Think C Build Library command to link the extension and write it to a library file.

Loading the Extension

Create a group file as described in Chapter 18, “Using The Director Translation Kit.” The group file’s loadable unit file name should point to the library generated by Think C.

Apple Macintosh PowerPC

Compiling an Extension

To create ScriptX extensions to use on the Macintosh, you must have CodeWarrior.

Note – The Macintosh versions of ScriptX and the KMP are compiled with XXXX of the CodeWarrior compiler.

For each extension, create a single source file and compile the module using the following switches:

See the printed documentation for current information.

Linking the Extension

See the printed documentation for current information.

Loading the Extension

See the printed documentation for current information.

Create a group file as described in Chapter , “Extending ScriptX.” The group file’s loadable unit file name should point to the library generated by Think C.

Microsoft Windows

See the printed documentation for current information.

Compiling an Extension

To create ScriptX extensions to use on Microsoft Windows, you must have Watcom C/386, V9.5.

Important – The Windows versions of ScriptX and the KMP are compiled using version 9.5 of the Watcom C compiler. Watcom has recently introduced version 10.0a of this product which may contain different internal symbols. Watcom doesn’t guarantee backward compatibility of code compiled using version 10.0a of the compiler. As a result, the ScriptX Loader may not work properly with code compiled with version 10.0a.

For each extension, you create a single source file and compile the module using the following switches:

```
wcc386p -mf -4s -w4 -zq -fpi -s -j -zpz -bt=windows foo.c
```

```
-mf           flat memory model
-4s           optimize for 486 and pass arguments on stack
-fpi          floating point math is done inline fpu instructions
-s            remove stack checks
-j            signed char by default
-bt=windows   build for Windows
-zpz          pack structures on word boundaries
```

Linking the Extension

Do not link the output of the compiler. ScriptX loads the object file (`.obj`) directly.

Loading the Extension

Create a group file as described in Chapter 9, “The ScriptX Loader.” The group file’s loadable unit file name should point to the `.obj` file output by the compiler.

OS/2

See the printed documentation for current information.

Compiling an Extension

To create ScriptX extensions to use with OS/2, you must have Watcom C/386, V10.0.

Important – The Windows versions of ScriptX and the KMP are compiled using version 10.0 of the Watcom C compiler. Watcom has recently introduced version 10.0a of this product which may contain different internal symbols. Watcom doesn’t guarantee backward compatibility of code compiled using version 10.0a of the compiler. As a result, the ScriptX Loader may not work properly with code compiled with version 10.0a.

For each extension, you create a single source file and compile the module using the following switches:

```
wcc386p -mf -4s -w4 -zq -fpi -s -j -zpz -bt=windows foo.c

-mf           flat memory model
-4s           optimize for 486 and pass arguments on stack
-fpi          floating point math is done inline fpu instructions
-s            remove stack checks
-j            signed char by default
-bt=windows   build for Windows
-zpz          pack structures on word boundaries
```

Linking the Extension

Do not link the output of the compiler. ScriptX loads the object file (`.obj`) directly.

Loading the Extension

Create a group file as described in Chapter 9, “The ScriptX Loader.” The group file’s loadable unit file name should point to the `.obj` file output by the compiler.

CHAPTER

21

Inter-Application Communication

ScriptX supports different forms of Inter-Application Communication on different platforms:

DDE on Windows and OS/2

AppleEvents on Macintosh and Power Macintosh

Dynamic Data Exchange (DDE)

ScriptX supports DDE for Microsoft Windows and OS/2. The Kaleida Media Player can operate as a DDE server, meaning that you can send it DDE commands from another application and it will respond. The following are the server name and commands for ScriptX:

`DDE servername:`	`"ScriptX"`
`DDE commands:`	`[FileOpen(`*filename*`)]` – open a document
	`[CompileSelect(`*filename*`)]` – Compile an ASCII script
	`[FilePrint(`*filename*`)]` – Print a ScriptX title
	`[QuitApp(`*filename*`)]` – Quit ScriptX

Apple Events

ScriptX supports the following Apple events. (ScriptX does not support AppleScript—it has no dictionary.) These were in version 1.0, but were not documented.

• Open application	(startup ScriptX)
• Open document	(startup ScriptX with a specific title)
• Print document	(print a ScriptX title)
• Quit application	(quit ScriptX)

ScriptX also supports the Apple event `Do Script`, to compile a ScriptX script.

Apple Event “Do Script”

ScriptX supports the `Do Script` Apple event only in the ScriptX development environment, and not in the Kaleida Media Player.

DISCLAIMER: We cannot guarantee that Do Script will work the same way or return the same values in future releases of ScriptX.

In general, Apple Events are a mechanism for one Macintosh application (the source) to communicate with another Macintosh application (the target). The `Do Script` Apple event enables the source application to send a script to ScriptX for execution, as follows.

In the source application, you assign a ScriptX script to a `Do Script` event and send the event to ScriptX. The `Do Script` event asks ScriptX to execute the expressions specified in its script.

When ScriptX receives a `Do Script` event, ScriptX wraps the associated script as a stream, spawns a separate thread and executes the `fileIn` method (defined in `ByteStream`) on the stream. `Do Script` returns nothing. The `fileIn` method compiles the stream into ScriptX bytecode and executes the results. As you would expect with `fileIn`, the script does not appear in the Listener, but the value of the last expression and any script errors do appear. When finished, it prints “Script done” in the Listener.

The following is the specification for the `Do Script` event.

Event Class	`kAEMiscStandards`	`'misc'`
Event ID	`kAEDoScript`	`'dosc'`
Parameter		
	`keyDirectObject`	
	Description:	The ScriptX script to execute. This can be any series of ScriptX expressions in text form. Length of text limited only by memory. (?)
	Descriptor type:	`typeIntlText`
	Required or Optional?	Required
Reply Parameter		
	nothing	

For more information on `Do Script`, see the *Apple Event Registry Standard Suites* document distributed by Apple (located in Dev.CD Mar 95:Technical Documentation: Apple Events Registry & Suites:Apple Events Registry page 9-404).

AppleScript Dictionary for “DoScript”

Implementing the AppleScript dictionary will allow a developer to execute a ScriptX expression from AppleScript, such as:

```
tell application “ScriptX”
   DoScript "new Window"
end tell
```

Note – The AppleScript Dictionary feature is not yet implemented.

Index

Symbols

<<< 71
<<<- 72
-- <<< 71
-- >>> 71
<eof> 71
>>> 71
\ 68

A

About Debugger menu command 111
About menu command 79, 90
About menu item 139
About ScriptX
  menu command 53, 77
actions
  classes of 213
Add Button 113
AddSpriteToStageAction class 215, 224
AIFF files
  importing 185
Apple Macintosh
  compiling, linking and loading 345
ASCII text
  importing 174
authoring
  diagram 65

B

backslash 68
batch importing 196
bitmaps
  importing 177
  importing from DIB files 178, 181
  importing from PICT files 177
  importing from QuickTime 183
breakpoint functions 104
breakpoints
  defining functions 104
  setting 103
Breakpoints window 113
Browse Item menu command 90
Browser menu 90
Browsers
  menu command 111
browsers 83
  API reference 90
  loading 83
  updating 84
Browsers menu command 80, 90
browsing
  see inspecting
building
  customized director imorters 262
bytecode compiler 65
bytecode instructions
  viewing in debugger 99
ByteCodeMethod Profiler 117

C

call stack 97
  menu 97
  viewing in debugger 97
cast translators 259
  building customized translators 265
  what they do 261
ChangeInkAction class 216, 226
ChangeSoundAction class 215, 227
ChangeSpriteAppearanceAction class 216, 229
ChangeSpriteSizeAction class 216, 230
ChangeTempoAction class 215, 231
Cinepak compressed movies
  importing 183
Class Browser 83
  loading 83
classes
  inspecting 83
Clear menu command 54, 78
clickHandler 138
Close Title menu command 53, 76
code
  entering in the Listener 68
code panel
  in debugger 99
Compile menu command 111
compiler 65
compiling
  Apple Macintosh C code 345
  Microsoft Windows C code 346, 347
  quickly 71, 72
  ScriptX scripts 70
  single-step 71
Components menu command 79
Cont Button 112
continuation character 68
Copy menu command 54, 78
creating
  ScriptX scripts 70
customized director importers
  building 262
Cut menu command 54, 78

D

debug global variable 334
Debugger 93, 95, 115, 123
  API reference 110
  breakpoint functions 104
  call stack menu 97
  code panel 99
  description 97
  entering immediately 100
  frame args panel 98
  loading 95
  opening 95
  setting breakpoints 103
  Step button 100
  Step Over button 100
  stepping through code 100
  viewing bytecode instructions 99
  viewing source code 99
  watchpoints 106
Debugger menu 110
Debugger menu command 80, 111
Debugger window 111
debugger.sxt 95
debugging 95
  exceptions 107
DebugInfo class 107
DeltaPathAction class 213
development process
  diagram 65
DIB files
  importing 178, 181
Director Importer
  introduction 199
  loading 204
Director titles
  importing 199
  preparing for importing 203
Director Translation Kit 257, 283
  loading 262
Director Translation Kit API 281
DirectorImporter class 232
Director-to-ScorePlayer Importer API Reference 221
DragRegion class 141
DTK
  see Director Translation Kit
DTK class 284
DTKBitmap class 286
DTKButton class 287
DTKCastMember class 288
DTKCastMemberToAudioStream class 289
DTKCastMemberToPresenter class 290
DTKCastMemberToStencil class 292
DTKCastTranslator class 294
DTKScoreFrame class 297
DTKScoreTranslator class 298
DTKShape class 300
DTKSound class 302
DTKSoundChannel class 303
DTKSpriteChannel class 304
DTKTempoChannel class 306
DTKText class 296, 307
DTKTransitionChannel class 309
DTKUnknown class 310
DTKVideo class 311

E

edit 134
Edit Button 113
Edit menu 54, 77
end of file marker 71
entryPoint function (External API) 335
exceptions
  debugging 107
executing
  more than one line in Listener 70
  previously entered line in Listener 69
Exit menu command 53, 73, 77
exitPoint function (External API) 335
exporting 171
  objects as media files 169
expressions
  incomplete 69

F

falseObject global constant 334
File menu 53, 75
fileIn instance method 65, 70, 75
Fonts menu command 79
frame args panel
  in debugger 98
free methods
  example 87

G

garbage collector ??–128
getMessage 136
Global instances
  theImportExportEngine 170
global variables
  theScratchTitle 73
  theScriptDir 65
Go Back button 84
Go Button 112
group file, see loadable groups

I

images
  importing 177
ImportDirector function 204
imported animations
  modifying 211
  reading the action list 217

timing control 210
importers
file organization 169
loading 59, 60
importing
AIFF 185
bitmaps 177
cinepak compressed key frames 183
DIB files 178, 181
Director titles 199
images 177
media files 169
MIDI files 190
movies 192
pict files 177
QuickTime 192, 194
QuickTime frames 183
RTF files 174
SND 186
sound 185
specifying source files 206
text 174
WAVE files 188
ImportMedia method
general syntax 170
importrs directory 59, 61, 169
incomplete expressions 69
inkmodes 216
inquire 134
inspect 134
inspect method 83
inspecting
classes 83
instances 83, 85
modules 83
installing
Macintosh version of KMP 50
OS/2 version of KMP 51
Windows version of KMP 50
Instance Browser 83
loading 83
instances
inspecting 83, 85
InterpolateAction class 213
Introduction 57
introduction 59
investigating exceptions 107
invoking
debugger immediately 100

K

Kaleida authoring player, see ScriptX Language and Class Library
Kaleida Image Compression (KIC) 163
Kaleida Media Player 47, 65
installing Macintosh version 50
installing OS/2 version 51
installing Windows version 50
starting 52
KIC plugins 163
KMP
see Kaleida Media Player

L

language tools 131
linking
Apple Macintosh C code 345
Microsoft Windows C code 346, 347
Listener 67
Listener menu command 76
Listener window 67
entering scripts 68
pasting text 73
loadable extensions 317—??
access to C data structures 332
common errors 333
entry point function 327
examples 322—333
exporting functions to the scripter 325—327
external API 333—??
external API header file 343—??
rules for creating 323
loadable groups 317—321
group file 318
processing 320
loadable units 318—322
processing 321
releasing 322
`relinquish` method (`LoadableUnitID`) 326
Loader component 317—322
loading
Apple Macintosh C code 345
browsers 83
Debugger 95
Director Importer 204
Director Translation Kit 262
importers 59, 60
Microsoft Windows C code 346, 347
ScriptX scripts 70
ScriptX scripts from menu 71
tools 60
LoopAction class 233

M

machine-specific notes 345
media
importing 169
media editors 131
memory management
Visual Memory 125
Memory Management component ??—128
`MemoryObject` class 323, 327, 341—342
`SXdereference` method 342

SXgetOSHandle method 342
SXlockMem method 342
SXreadAt method 342
SXunlockMem method 342
SXwriteAt method 342
menu commands
About 79, 90
About Debugger 111
About ScriptX 53, 77
Back 92
Browsers 80, 90, 111
Clear 54, 78
Close 76
Close Title 53, 76
Compile 110, 111
Components 79
Copy 54, 78
Cut 54, 78
Debugger 80, 111
Edit 92
Exit 53, 73, 77
Interrupt 110
Listener 76
New Listener 73
Open Accessory 53, 76
Open Title 53, 75
Open title 71
Page Setup 53, 77
Paste 54, 78
Picklist 91
Preferences 79, 90, 111
Print 53, 77
Print Setup 53, 77
Quit 53, 73, 77, 79, 90, 111
Refresh 84, 92, 110
Remove 110
Resume Callback 110
Select All 79
Select IVS by 91
Set Breakpoint 92, 103
Show 110
Sort order 90
Specialization level 91
Undo 54, 78
Visual Memory 76
menus
Browser 90
Debugger 110
Edit 54, 77
File 53, 75
Tools 79, 90, 111
tool-specific 80
Window 54, 79
metaphor tools 131
method profiler 117
Microsoft Windows
compiling, linking and loading 345, 346, 347
MIDI files
importing 190
modifying
imported animations 211
modules
inspecting 83
movies
importing 192

N

New Debugger on Exception preference 95
New Listener menu command 73, 76

O

Open Accessory menu command 53, 76
Open Title menu command 53, 70, 71, 75
opening
debugger 95
ScriptX scripts 70
ScriptX scripts from menu 71

P

Page Setup menu command 53, 77
Paste menu command 54, 78
pasting text
from the Listener 73
PathAction class 214
Photoshop KIC plugins 163
Pict files
importing 177
platform-specific notes 345
plugins
kic for Photoshop 163
preferences
New Debugger on Exception 95
Preferences menu command 79, 90, 111
Preferences menu item 139
PrepareForTransitionAction class 234
PrepareTransitionAction class 217
Primitive class 325, 326
Print menu command 53, 77
Print Setup menu command 53, 77
Profiler 117

Q

quick-compiling 71, 72
QuickTime files
importing 192, 194
importing as bitmaps 183
Quit menu command 53, 73, 77, 79, 111

R

Refresh menu command 84, 110
Refresh Window menu command 91, 92

Remove Button 113
Remove menu command 110
RemoveSpriteFromStageAction 236
RTF files
  importing 174

S

sample scripts 73
Save Change menu command 91
Score class 237
score translators 259
  building customized translators 268
  what they do 262
ScorePlayer class 239
ScorePlayerCastTranslator class 242
ScorePlayerScoreTranslator class 244
ScoreTicker class 245
script files
  compiling 70
  creating 70
  loading 70
  loading from menu 71
  sample scripts 73
ScriptAction class 214
scripts
  entering in the Listener 68
ScriptX
  starting 66
ScriptX Developer's CD 73
ScriptX Listener and Menus 25, 63
ScriptX Listener window 67
ScriptX scripts
  entering in the Listener 68
Select All menu command 79
sendMessage 136
Set Breakpoint menu command 103
Show Class Methods menu command 92
Show Instance Methods menu command 92
Show Instance Variables menu command 92
Show menu command 110
showChangedRegion instance variable 65
single-step compiling 71
SND
  importing 186
sound
  importing 185
source code
  stepping through 100
  viewing in debugger 99
source files
  for importing 206
Sprite class 246
SpriteChannelInfo class 247
starting Kaleida Media Player 52
starting ScriptX 66
Step Button 100, 112
Step Over Button 100, 112
stepping
  through code 100
`SXbool` type 334
`SXdereference` method (`MemoryObject`) 342
`SXdouble` type 334
`SXdoubleFrom` function (External API) 336
`SXdoubleToObject` function (External API) 335
`SXempty` global constant 334
`SXextCall` function (External API) 336
`SXextGetGlobal` function (External API) 337
`SXextGetIV` function (External API) 337
`SXextGetModule` function (External API) 338
`SXextMakeFunction` function (External API) 338
`SXextMakeGeneric` function 327
`SXextMakeGeneric` function (External API) 338
`SXextMakeGlobal` function (External API) 339
`SXextSetGlobal` function (External API) 340
`SXextSetIV` function (External API) 340
`SXgetOSHandle` method (`MemoryObject`) 342
`SXint` type 334
`SXintFrom` function (External API) 336
`SXintToObject` function (External API) 335
`SXlockMem` method (`MemoryObject`) 342
`SXmakeMemoryObject` function (External API) 341
`SXmakeNameInterned` function (External API) 339
`SXMakePrimitive` function 326
`SXmakeString` function (External API) 335
`SXobject` type 334
`SXreadAt` method (`MemoryObject`) 342
`SXreadStream` function (External API) 340
`SXstringOf` function (External API) 336
`SXthreadIdle` function (External API) 336
`SXthreadYield` function (External API) 336
`SXundefined` global constant 334
`SXunlockMem` method (`MemoryObject`) 342
`SXunsupplied` global constant 334
`SXwriteAt` method (`MemoryObject`) 342
`SXwriteStream` function (External API) 340
`SXwriteString` function (External API) 340

T

TargetListAction class 214
text
  importing 174
TextSprite class 248
The Browser System 81
theImportExportEngine global instance 205
theImportExportEngine global variable 170
theScratchTitle global variable 73
theScriptDir global variable 65
theToolOrganizer 134
Think C compiler 345
title tools 132
tlfrmwrk.lib 95
Tool Framework 129
  implementation 133
tool organizer 134

tools
  loading 60
tools directory 60, 95
Tools Framework 60
Tools menu 79, 90, 111, 139
toolsx directory 60, 95
`trueObject` global constant 334
TwoDSprite class 250

U

Undo menu command 54, 78
updating
  browsers 84
user focus 74
Using The Director Translation Kit 255
Using the Director-to-ScorePlayer Importer 197

V

VideoSprite class 252
viewing
  see inspecting
Visual Memory 125
Visual Memory menu command 76

W

WaitForSoundAction class 217, 253
wantsClick 138
watchpoints
  using in debugger 106
Watchpoints window 113
Watcom C/ 386 compiler 345
WAVE files
  importing 188
widgets.sxl 95
Window menu 54, 79
windows
  Breakpoints 113
  Debugger 111
  Watchpoints 113
writing a tool in ScriptX 132

Colophon

PRODUCT DEVELOPMENT

VP Engineering • Chris Jette

Chief Architect • John Wainwright

Kaleida Founder • Erik Neumann

Kaleida Fellow • Andrew Nicholson

ScriptX Language Team • Wade Hennessey (mgr), Mike Agostino, Eric Benson, Ross Nelson, Chris Richardson, David Williams

ScriptX Media Team • Erik Neumann (mgr), Vidur Apparao, Ikko Fushiki, Jennifer Jacobi, Chih Chao Lam, Michael Papp, Ken Tidwell, Ken Wiens

Cross-Platform Team • Elba Sobrino (mgr), Yukari Huguenard, Alan Little, Jeanne Mommaerts, Charlie Reiman, Richard Roth, Vladimir Solomonik, Clayton Wishoff, Wanmo Wong

Quality Engineering Team • Ermalinda Horne (mgr), William Africa, Adela Bartl, Ron Decker, Suzan Ehdaie, Rajiv Joshi, Tony Leung

Technical Publications Team • Douglas Kramer (mgr), Jocelyn Becker, Alta Elstad, Maydene Fisher, Howard Metzenberg, Sandra Ware

Application Support Engineering Team • Ray Davis, Rob Lockstone, Felicia Santelli, Su Quek

AND ALL OUR FELLOW KALEIDANS

Masumi Abe, Harvey Alcabes, Rob Barnes, Amy Benesh, Fred Benz, Alison Booth, Mike Braun, Mark Bunzel, Janet Byler, John Cummerford, Shannon Garrow, Marylis Garthoeffner, Norman Gilmore, Bill Grotzinger, Sue Haderle, Diana Harwood, Don Hopkins, Bill Howell, John Hudson, Pat Ladine, Fritzi Lareau, Deb Lyons, Karl May, Steve Mayer, Victor Medina, Gabe Mont-Reynaud, Tom Morton, Randy Nelson, Christy O'Connell, Karen O'Such, Christian Pease, David Rosnow, Molly Seamons, Ken Smith, Michelle Smith, Ivan Vazquez, Greg Womack

THANKS TO KALEIDA ENGINEERING ALUMNI, INCLUDING:

Sarah Allen, Dan Bornstein, Jim Inscore, David Kaiser, Shel Kaphan, Laura Lemay, Dave Lundberg, Leslie Lundquist, Fred Malouf, Dmitry Nasledov, Steve Riggins, Steve Shaw, Cheng Tan, Phil Taylor

Special Thanks To...

Lady, Nikki, Boots, Ella, Tyler,Rufus, Kiri, Frisky,and Iggy

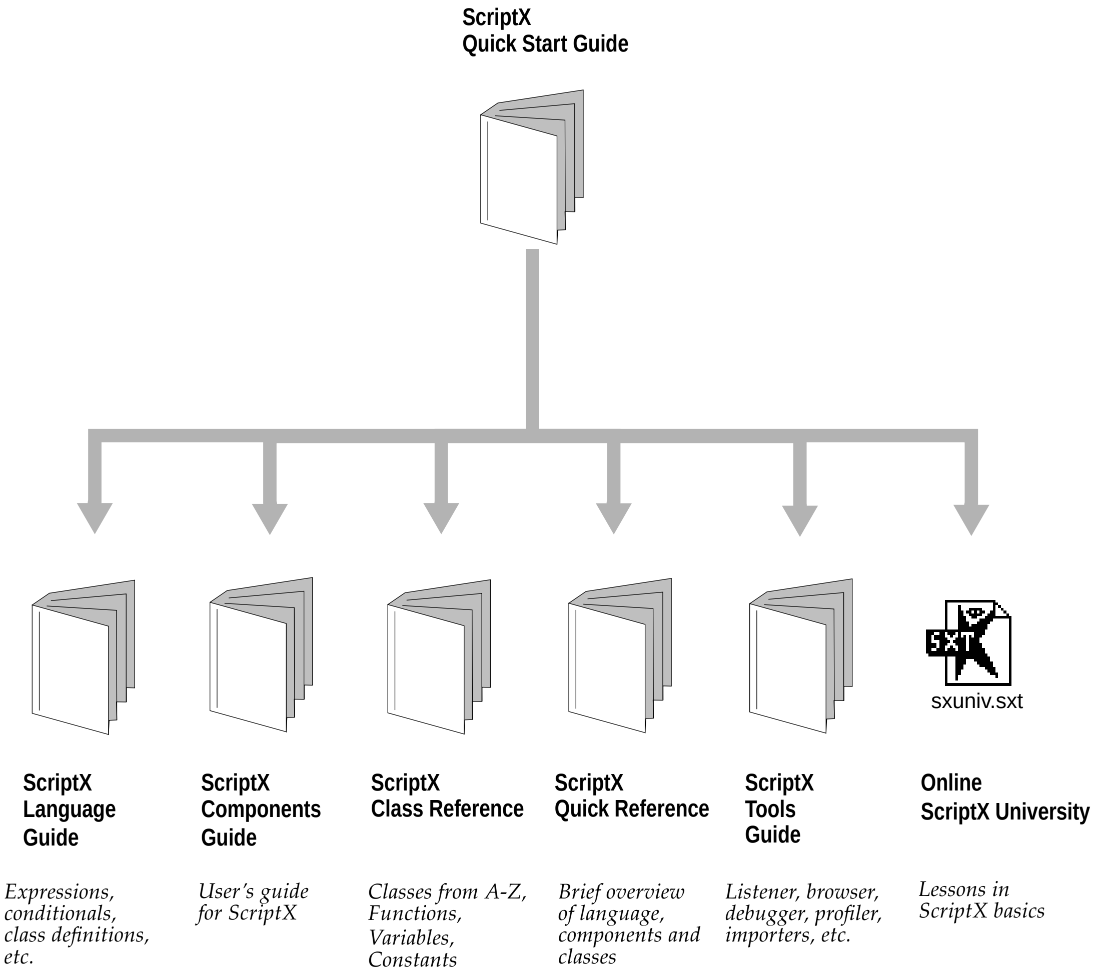
THIS DOCUMENT

Writing • Jocelyn Becker, Doug Kramer, Howard Metzenberg

Book production • Sandra Ware, Jacki Dudley, Diana Harwood, Beth Delson

This book was created electronically using Adobe FrameMaker on Macintosh Quadra computers.

Documentation Roadmap



ScriptX Quick Start Guide

ScriptX Language Guide

Expressions, conditionals, class definitions, etc.

ScriptX Components Guide

User's guide for ScriptX

ScriptX Class Reference

Classes from A-Z, Functions, Variables, Constants

ScriptX Quick Reference

Brief overview of language, components and classes

ScriptX Tools Guide

Listener, browser, debugger, profiler, importers, etc.

sxuniv.sxt

Online ScriptX University

Lessons in ScriptX basics

Acrobat versions are available for most manuals