Autodesk®
Visual Effects and Finishing
2010 Edition

Autodesk® Developer Network Sparks® API Reference Guide

Autodesk®

Autodesk® Inferno®, Autodesk® Flame®, Autodesk® Flint®, Autodesk® Fire®, Autodesk® Smoke®, Autodesk® Backdraft® Conform software

Portions relating to MXF-SDK was developed by Media, Objects and Gadgets - Soluçoes de Software e Hardware, S.A. (http://www.mog-solutions.com) in co-operation with Institut für Rundfunktechnik GmbH (http://www.irt.de).

Portions powered by Automatic Duck. © 2006 Automatic Duck, Inc. All rights reserved.

Portions relating to "dslib" C/C++ Copyright 1988-1989 Eugene Dronek and Rich Morin.

Autodesk® Flare™ software

Portions relating to MXF-SDK was developed by Media, Objects and Gadgets - Soluçoes de Software e Hardware, S.A. (http://www.mog-solutions.com) in co-operation with Institut für Rundfunktechnik GmbH (http://www.irt.de).

Portions powered by Automatic Duck. © 2006 Automatic Duck, Inc. All rights reserved.

Portions relating to "dslib" C/C++ Copyright 1988-1989 Eugene Dronek and Rich Morin.

Portions relating to MPEG Layer-3; supply of this product does not convey a license under the relevant intellectual property of Thomson multimedia and/or Fraunhofer Gesellschaft nor imply any right to use this product in any finished end user or ready-to-use final product. An independent license for such use is required. For details, please visit http://www.mp3licensing.com.

Autodesk® SystemCentral™ software

Adobe® Flash® Player. Copyright © 1996-2006 Adobe Systems Incorporated. All Rights Reserved.

Autodesk® Inferno®, Autodesk® Flame®, Autodesk® Flint®, Autodesk® Smoke®, Autodesk® Backdraft® Conform

Portions relating to MPEG Layer-3; supply of this product does not convey a license under the relevant intellectual property of Thomson multimedia and/or Fraunhofer Gesellschaft nor imply any right to use this product in any finished end user or ready-to-use final product. An independent license for such use is required. For details, please visit http://www.mp3licensing.com.

The following are registered trademarks or trademarks of Autodesk, Inc., in the USA and other countries: 3DEC (design/logo), 3December, 3December.com, 3ds Max, ADI, Alias, Alias (swirl design/logo), AliasStudio, Alias|Wavefront (design/logo), ATC, AUGI, AutoCAD, AutoCAD Learning Assistance, AutoCAD LT, AutoCAD Simulator, AutoCAD SQL Extension, AutoCAD SQL Interface, Autodesk, Autodesk Envision, Autodesk Insight, Autodesk Intent, Autodesk Inventor, Autodesk Map, Autodesk MapGuide, Autodesk Streamline, AutoLISP, AutoSnap, AutoSketch, AutoTrack, Backdraft, Built with ObjectARX (logo), Burn, Buzzsaw, CAiCE, Can You Imagine, Character Studio, Cinestream, Civil 3D, Cleaner, Cleaner Central, ClearScale, Colour Warper, Combustion, Communication Specification, Constructware, Content Explorer, Create>what's>Next> (design/logo), Dancing Baby (image), DesignCenter, Design Doctor, Designer's Toolkit, DesignKids, DesignProf, DesignServer, DesignStudio, Design|Studio (design/logo), Design Web Format, Discreet, DWF, DWG, DWG (logo), DWG Extreme, DWG TrueConvert, DWG TrueView, DXF, Ecotect, Exposure, Extending the Design Team, Face Robot, FBX, Filmbox, Fire, Flame, Flint, FMDesktop, Freewheel, Frost, GDX Driver, Gmax, Green Building Studio, Heads-up Design, Heidi, HumanIK, IDEA Server, i-drop, ImageModeler, iMOUT, Incinerator, Inferno, Inventor, Inventor LT, Kaydara, Kaydara (design/logo), Kynapse, Kynogon, LandXplorer, LocationLogic, Lustre, Matchmover, Maya, Mechanical Desktop, Moonbox, MotionBuilder, Movimento, Mudbox, NavisWorks, ObjectARX, ObjectDBX, Open Reality, Opticore, Opticore Opus, PolarSnap, PortfolioWall, Powered with Autodesk Technology, Productstream, ProjectPoint, ProMaterials, RasterDWG, Reactor, RealDWG, Real-time Roto, REALVIZ, Recognize, Render Queue, Retimer,Reveal, Revit, Showcase, ShowMotion, SketchBook, Smoke, Softimage, Softimage|XSI (design/logo), SteeringWheels, Stitcher, Stone, StudioTools, Topobase, Toxik, TrustedDWG, ViewCube, Visual, Visual Construction, Visual Drainage, Visual Landscape, Visual Survey, Visual Toolbox, Visual LISP, Voice Reality, Volo, Vtour, Wire, Wiretap, WiretapCentral, XSI, and XSI (design/logo).

The following are registered trademarks or trademarks of Autodesk Canada Co. in the USA and/or Canada and other countries: Backburner, Multi-Master Editing, River, and Sparks.

The following are registered trademarks or trademarks of Moldflow Corp. in the USA and/or other countries: Moldflow MPA, MPA (design/logo), Moldflow Plastics Advisers, MPI, MPI (design/logo), Moldflow Plastics Insight, MPX, MPX (design/logo), Moldflow Plastics Xpert.

Disclaimer

Published by:

Autodesk, Inc.

111 Mclnnis Parkway

San Rafael, CA 94903, USA

Title:	Autodesk Visual Effects and Finishing 2010 Edition Autodesk Developer Network Sparks Api Reference Guide
Document Version:	1
Date:	March 18, 2009

Contents

1 Introduction

Summary

About Sparks

A Sparks® plug-in defines an interactive user interface similar to any of the standard modules in Autodesk® Visual Effects and Finishing products and can be used to load clips from the desktop or EditDesk, manipulate them, and process new clips back to the desktop or EditDesk. These plug-ins can be used to provide image and audio processing features that are not already available with Autodesk® Inferno®, Autodesk® Flame®, Autodesk® Flint®, Autodesk® Flare®, and Autodesk® Smoke®.

In Autodesk Visual Effects products, some Sparks can be used in Batch from the Spark node. In Autodesk Finishing products, some Sparks can be used as soft effects.

Sparks plug-ins allow you to customize your environment and expand the capabilities of your system.

After you create a Sparks plug-in, you can copy it to the */usr/discreet/sparks* directory of a Autodesk Visual Effects and Finishing product. For information on how to load a Sparks plug-in into a Autodesk Visual Effects and Finishing product, refer to the "Sparks" chapter in your application user guide.

Compatibility

All the Sparks plug-in conventions detailed in this guide are compatible with Inferno 2010, Flint 2010, Flame 2010, Flare 2010, and Smoke 2010.

System Requirements

You can use the C++ compiler that comes with your Linux® operating system to build Sparks.

Files related to Sparks plug-ins are located in the *~/sparks* directory. They include the following.

***Makefile* —** This file includes parameters and code for compiling and linking Spark code. Once compiled and linked, Sparks are loaded onto Autodesk Visual Effects and Finishing products.

***spark.h* —** The header file that must be included in the Sparks plug-in. It contains all the function declarations required to compile a Sparks DSO (dynamic shared object).

***spark*.c* —** Sample Sparks plug-in files distributed with Autodesk Visual Effects and Finishing products.

When compiling, you must use the *spark.h* header file distributed with your Autodesk Visual Effects and Finishing product. If you use a more recent header, you might have unresolved symbols when linking the Sparks plug-in to your Autodesk Visual Effects and Finishing product.

The Sparks API is maintained for upwards compatibility, but new versions of the API introduce new functions. If you use these new functions when developing your Sparks plug-ins, be aware that users who have not upgraded their systems will not be able to run your Sparks plug-ins.

Using This Guide

The *Autodesk Sparks API Reference Guide* has been created to allow third parties to develop software plug-ins for use with Autodesk Visual Effects and Finishing products. This guide describes:

- The Sparks interface
- Functions required by a Sparks plug-in
- The organization of the Sparks user interface
- Utility functions that are provided in the API including audio functions

In addition to reading this document, Sparks developers should examine the sample Sparks source files supplied with Autodesk Visual Effects and Finishing products. You can find these files in the *~/sparks* directory.

Related Documentation

Read this document in conjunction with the documentation set of your Autodesk Visual Effects and Finishing product. Refer to the "Sparks" chapter in your application user guide.

You can also learn about Linux configuration in the *Linux Configuration Guide for Autodesk Sparks Plug-in Developers* available through the Autodesk Developer Network Sparks program.

Documentation Conventions

The following notation conventions are used in this document.

Convention	Example
Italics denote UNIX files and directories.	*/usr/project1*
Sparks functions appear in Courier bold font.	**`SparkInitialise`**
Output from functions is printed in Courier regular font.	`SPARK_FLAG_Y`

Getting Help

If you need more information or help with the Sparks Plug-In API, contact the Autodesk Developer Network (ADN) Sparks program: http://www.autodesk.com/joinadn

2 Sparks API

Summary

About the Sparks API

A Sparks plug-in is a dynamic shared object (DSO). This is a compiled object that can be dynamically loaded and linked to a process at run time.

Each Sparks plug-in uses a standard internal format. Since Autodesk Visual Effects and Finishing products makes calls to the Sparks routines, you must correctly name and implement certain functions in order to create a working Sparks plug-in.

Mandatory—SparkInfoStruct

The parameter of type **`SparkInfoStruct`** appears in many of the functions. This parameter defines the interface between Autodesk Visual Effects and Finishing products and the Sparks environment, and also contains information required for processing source images. The structure type is defined in the *spark.h* header file, which must be included in each Sparks source file.

Sparks Interface Functions

The following section provides the correct naming for mandatory, optional, and recommended interface functions that you can use to facilitate communication between Autodesk Visual Effects and Finishing products and Sparks plug-ins.

NOTE: The first four functions are mandatory.

Mandatory—SparkInitialise ( )

```
unsigned int SparkInitialise ( SparkInfoStruct SparkInfo )
```

Autodesk Visual Effects and Finishing products call the **SparkInitialise** function in three situations:

- When you invoke a new Sparks plug-in.

NOTE: When you use the same Sparks plug-in numerous times, the **SparkInitialise** function is only called the first time.

- When the resolution of the input used with the Sparks plug-in changes (in Batch or on the desktop).
- When the resolution of the output of the Sparks plug-in changes. (In this case, the Sparks plug-in has no input.)

A Sparks plug-in typically uses this function to perform the following:

- To load a setup that was saved from the last invocation of the Sparks plug-in and that sets the user interface variables to the previously defined values.
- To allocate any resources, such as dynamic memory, that the Sparks plug-in requires.

The return value of the **SparkInitialise** function determines whether the Sparks plug-in operates at the desktop level or at the component level, and it is defined by the "bit-wise or" of the required levels. For example, if the Sparks plug-in is to operate at both the desktop and component levels, the return value would be `(SPARK_DESKTOP | SPARK_MODULE)`. In this case, it is up to you to supply software that operates at both these levels. See "Desktop and Component Levels" on page 21.

Mandatory—SparkClips( )

```
int SparkClips( void )
```

The **SparkClips** function tells Autodesk Visual Effects and Finishing products how many source clips are required by the Sparks plug-in. For example, a Sparks plug-in used for image compositing requires a front clip, back clip, and matte clip in order to produce an output clip. Therefore, the **SparkClips** function would return a value of `3`. A Sparks plug-in used for image blending requires only two source clips; in this case, the **SparkClips** function would return a value of `2`.

Autodesk Visual Effects and Finishing products use the return value to determine which of the Front, Back, and Matte control buttons appear at the component level. See "The Component Level" on page 21.

Mandatory—SparkProcess( )

```
unsigned long *SparkProcess( SparkInfoStruct SparkInfo )
```

The **SparkProcess** function contains the processing algorithms (written by you) of the Sparks plug-in. The function is called once for every output frame to be generated, as indicated by the return value of the **SparkProcessStart** function. The **SparkProcess** return value must be the address of the image buffer that contains the processed image. Autodesk Visual Effects and Finishing products use the contents of the image buffer to generate a frame of the output clip on the destination reel (or EditDesk); the destination reel is selected when the user calls the Sparks plug-in from the desktop. If the return value is NULL, no action is taken by the calling environment of the Autodesk Visual Effects and Finishing products.

Mandatory—SparkUnInitialise( )

```
void SparkUnInitialise( SparkInfoStruct SparkInfo )
```

The **SparkUnInitialise** function can be used to save user interface configurations or to release any resources that were allocated by the **SparkInitialise** function. There is no associated return value for this function. It is called in the following situations:

- When another Sparks plug-in is invoked from the desktop.
- When the resolution of the input used with the Sparks plug-in changes (in Batch or on the desktop).
- When the resolution of the output of the Sparks plug-in changes. (In this case, the Sparks plug-in has no input.)
- Upon exit of Autodesk Visual Effects and Finishing products.

SparkProcessEnd( )

```
void SparkProcessEnd( SparkInfoStruct SparkInfo )
```

SparkProcessEnd( ) is executed once at the end of the processing loop whether it is completed or aborted.

NOTE: SparkProcess( ) must return a valid pointer to a frame buffer if the Sparks plug-in is called from Batch.

SparkProcessStart( )

```
int SparkProcessStart( SparkInfoStruct SparkInfo )
```

SparkProcessStart( ) is no longer mandatory. If not defined, it is assumed to return 1. The **SparkProcessStart** function determines the frame layout of the processed clip. The return value of the function indicates the number of frames to be generated for the current input frame. Since in most cases there is a one-to-one relationship between the length of the input clip and the length of the output clip, the function return value is usually 1. The value is different than 1 for functions in which the length of the processed clip is not the same as that of the source clip, for example, when using a timewarp function in which the source clip is either expanded or compressed in time. For an example, see the file *sparkAverage1.c*, located in the *~/sparks* directory.

NOTE: A Sparks plug-in can also request any frame of any input clip using the **sparkGetFrame** utility function. See Chapter 3, "Sparks Utility Library" on page 27.

Recommended—SparkMemoryTempBuffers( )

```
void SparkMemoryTempBuffers( void )
```

The **SparkMemoryTempBuffers** function (introduced with Inferno 2.5, Smoke 2.5, Flame 5.5, and Flint 5.5 [Sparks API version 2.5+]) allows you to access the memory buffer interface. This function, unlike the memory management interface from previous versions, allows memory to be allocated for loading Sparks plug-ins on-the-fly, and is essential to batch processing. The API memory functions used to register image buffers have an effect only through this function.

NOTE: There is no return value associated with this function.

HINT: The new memory management system, which is more resource-friendly and provides greater stability is recommended. It is required to load a Sparks plug-in from Batch or to access the Player from a Sparks plug-in. See "Memory Buffer Management" on page 13.

SparkEvent( )

```
void SparkEvent( SparkModuleEvent Event )
```

The **SparkEvent** function can be used to notify Sparks plug-ins of certain user events (defined in *spark.h* header file).

SparkSetupIOEvent( )

```
void SparkSetupIOEvent( SparkModuleEvent Event, char *Path,
char *File)
```

The **SparkSetupIOEvent** function can be used to notify a Sparks plug-in of user load or save setup events. The path and file parameters will receive the values entered by the user.

SparkIdle( )

```
void SparkIdle( void )
```

The **SparkIdle** function can be used to determine when the application is idle and waiting for an event.

SparkFrameChange( )

```
void SparkFrameChange( SparkInfoStruct SparkInfo )
```

The **SparkFrameChange** function can be used by Sparks plug-ins that need to be notified of frame changes.

SparkAnalyse( )

```
ulong * SparkAnalyse( SparkInfoStruct SparkInfo )
```

SparkAnalyse( ) works like **SparkProcess( )** but does not generate any output clips.

The field **SparkInfo.TotalFrameNo** passed to this function gives the maximum video length among the input clips. If none of the input clips contain video frames, **SparkInfo.TotalFrameNo** is set to `1`. **SparkInfo.TotalFrameNo** is defined in the *spark.h* header file.

SparkAnalyseStart( )

```
int SparkAnalyseStart( SparkInfoStruct SparkInfo )
```

SparkAnalyseStart( ) is used to control the length ratio between the input and the output just as **SparkProcessStart( )** does. If not defined, it is assumed to return `1`.

The field **SparkInfo.TotalFrameNo** passed to this function gives the maximum video length among the input clips. If none of the input clips contain video frames, **SparkInfo.TotalFrameNo** is set to `1`. **SparkInfo.TotalFrameNo** is defined in the *spark.h* header file.

SparkAnalyseEnd( )

```
void SparkAnalyseEnd( SparkInfoStruct SparkInfo )
```

SparkAnalyseEnd( ) is executed once at the end of the analysis loop whether it is completed or aborted.

The field **SparkInfo.TotalFrameNo** passed to this function gives the maximum video length among the input clips. If none of the input clips contain video frames, **SparkInfo.TotalFrameNo** is set to 1. **SparkInfo.TotalFrameNo** is defined in the *spark.h* header file.

SparkInteract( )

```
unsigned long *SparkInteract( SparkInfoStruct SparkInfo, int
SX, int SY, float Pressure, float VX, float VY, float VZ )
```

The **SparkInteract** function provides for interactive image manipulation by defining the Sparks actions when a mouse, tablet, or other input device event occurs in the image window. For example, this function can be used to define a processing region or crop box in the image.

The input parameters are used as follows:

- SX and SY correspond to the screen coordinates of the input device event, with the origin defined as the lower-left corner of the screen.
- Pressure indicates the pressure exerted on the input device. The range of values for this parameter is from 0 to 1.
- VX and VY correspond to image coordinates, with the origin defined as the lower-left corner of the image. Note that the current position of the image origin in screen coordinates is given by the **FrameBufferX** and **FrameBufferY** fields of the **SparkInfo** structure.
- VZ is the zoom factor.

The return value should be the address of an image buffer used to update the image window. If the return value is NULL, no action is taken by the calling environment.

SparkOverlay( )

```
void SparkOverlay( SparkInfoStruct SparkInfo, float
ZoomFactor )
```

The **SparkOverlay** function provides a way to add user-defined information on top of the image window that is not actually part of the image. Typically, the function is used to draw things such as object handles, axes, or crop boxes. The function is called every time the image window is redrawn, immediately after refreshing the image itself.

WARNING: Sparks writers should use the **SparkOverlay** function carefully, making sure to restore the graphical context to its original state before exiting the function. Autodesk Visual Effects and Finishing products cannot prevent a Sparks plug-in from changing graphical parameters required for normal operation. If Autodesk Visual Effects and Finishing products start behaving strangely after running a Sparks plug-in that uses the **SparkOverlay** function, the

first debugging step should be to comment out **SparkOverlay** from that Sparks plug-in and run the Autodesk Visual Effects and Finishing product again.

SparkChannelEditor( )

void SparkChannelEditor(void)

The **SparkChannelEditor** function lets you customize the Channel Editor layout by creating channel hierarchies. It is invoked by the Autodesk Visual Effects and Finishing product before the **SparkInitialise** function. The **sparkCeAddFolder** and **sparkCeAddControl** API functions have an effect only through this function.

SparkIsInputFormatSupported( )

int SparkIsInputFormatSupported(SparkPixelFormat fmt)

The **SparkIsInputFormatSupported** function allows a Spark to tell IFF whether it supports a given pixel format.

The input parameter **fmt** must be one of the supported pixel formats in the **SparkPixelFormat** enumeration (declared in the *sparks.h* header file).

This function must return 1 if the Spark supports the given format. Otherwise, it should return 0. If a Spark does not provide this function, support for the SPARKBUF_RGB_24_3x8 and SPARKBUF_RGB_48_3x12 formats is assumed, thus maintaining API compatibility with existing Sparks.

Note that support for certain formats is subject to the license restrictions of Autodesk's Visual FX and Finishing applications. A Spark that supports RGB 16-bit floating point format may not be allowed to process an image in that format if the application (into which the image is loaded) does not support 16-bit floating point processing.

The following image conversion and processing functions only support SPARKBUF_RGB_24_3x8 and SPARKBUF_RGB_48_3x12 images:

`sparkMonochrome()`	`sparkNegative()`
`sparkFromHLS()`	`sparkToHLS()`
`sparkFromYUV()`	`sparkToYUV()`
`sparkBlur()`	`sparkComposite()`

Calling Sequence for Sparks Interface Functions

This section shows the sequence in which Autodesk Visual Effects and Finishing products call the Sparks interface functions. For the sake of simplicity, the function parameters are not shown.

New Memory Model

```
clip_total = SparkClips( )
SparkMemoryTempBuffers( )
SparkChannelEditor( )
SparkInitialise( )
for( i = 0; i < length_of_longest_input_clip; i++ ) {
  do {
    frame_total = SparkProcessStart( )
  } while( frame_total == 0 )
  for( j = 0; j < frame_total; j++ ) {
    result_buffer = SparkProcess( )
  }
}
SparkProcessEnd( )
SparkUnInitialise( )
```

Old Memory Model (Inferno/Smoke 2.5, Flame/Flint 5.5)

```
SparkChannelEditor( )
SparkInitialise( )
clip_total = SparkClips( )
for( i = 0; i < length_of_longest_input_clip; i++ ) {
  do {
  frame_total = SparkProcessStart( )
  } while( frame_total == 0 )
  for( j = 0; j < frame_total; j++ ) {
    result_buffer = SparkProcess( )
  }
}
SparkProcessEnd( )
SparkUnInitialise( )
```

Since the **SparkInteract** function is a ghost function that is called each time an input device event occurs, it is not shown in the calling sequence.

NOTE: The **SparkOverlay** function is also called whenever the image window needs refreshing.

Memory Buffer Management

There are two main ways to approach memory buffer management, referred to here as the old memory interface and the new memory interface. To maintain backwards compatibility with the Sparks plug-ins coded for versions earlier than Inferno 2.5, Smoke 2.5, Flame 5.5, and Flint 5.5, the Sparks API will default to the old memory buffer interface. The advantage of the new memory interface is a more flexible interface that includes the ability to use the Sparks plug-in from the Batch module.

Old Memory Interface

In general, a Sparks plug-in uses a set of `n` source clips, where `n` is determined by the return value of the **SparkClips** function, to produce one or more output frames. Autodesk Visual Effects and Finishing products pass the addresses of the input clips to the Sparks plug-in from

the **Buffers** field of the **SparkInfoStruct** input parameter. If multiple frames from the source clips are required to produce a single output frame, all but the last frame of information must be saved in temporary buffers. This can easily be achieved using the supplemental buffers that are provided in the **Buffers** field of **SparkInfoStruct**.

The maximum number of image buffers available to the Sparks plug-in is 17 plus the number of buffers declared on the Memory line of the configuration file. There is a restriction that the first `n` buffers are reserved for input from Autodesk Visual Effects and Finishing products. The additional buffers can be used in any fashion, and the address of the output image is returned to the calling environment as the return value of either the **SparkProcess** or the **SparkInteract** function.

If you require more image buffers than Autodesk Visual Effects and Finishing products provide, you may allocate them yourself. You must adhere to the following restrictions to allow the buffers to be passed back to Autodesk Visual Effects and Finishing products for display and/or processing:

- The buffers must be longword aligned in memory. Autodesk Visual Effects and Finishing products reject any buffer that does not follow this constraint.
- The buffer definition must correspond to one of the structures defined in the *spark.h* header file.

New Memory Buffer Interface

Sparks plug-ins access the memory buffer interface by defining the **SparkMemoryTempBuffers** function. When this function is defined, Autodesk Visual Effects and Finishing products will allocate `n` input buffers returned by the **SparkClips** function, plus one buffer for the result. Any extra buffer(s) must be registered within the **SparkMemoryTempBuffers** function's scope using one of the three memory functions of the Sparks Utility Library.

All Sparks plug-in memory buffers registered to Autodesk Visual Effects and Finishing products receive a unique ID number:

- The result clip's ID is `1`.
- The input clips receive IDs `2` to `n`.
- Any extra buffer registered use an ID from *n+1* to *n+x*.

These IDs are used by your processing routines to access specific memory buffers. The following example shows how you can use these IDs.

```
/* Aliases for Image Buffers */
static int RESULT_ID = 1;
static int AUX1_ID;
static int AUX2_ID;
static int AUX3_ID;
static SparkMemBufStruct SparkResult;
static SparkMemBufStruct SparkAux1;
static SparkMemBufStruct SparkAux2;
static SparkMemBufStruct SparkAux3;
void SparkMemoryTempBuffers( void )
{
  AUX1_ID = sparkMemRegisterBuffer( );
  AUX2_ID = sparkMemRegisterBufferSize(
  sizeof(unsigned long)*400 );
  AUX3_ID = sparkMemRegisterBufferFmt(
  200, 100, SPARK_FMT_MONO );
}
/* ...etc */
```

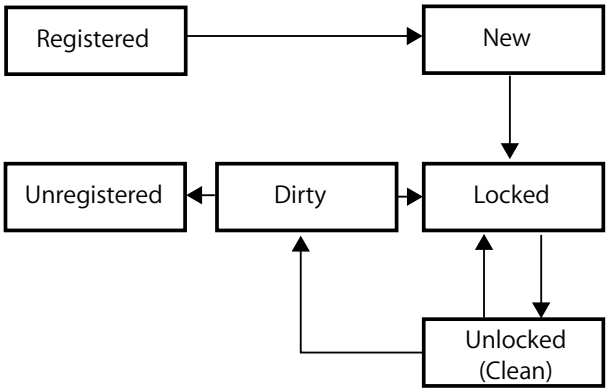
State Transition of Sparks Memory Buffers

Once a Sparks memory buffer is registered, Autodesk Visual Effects and Finishing products are in control of its state transition. Memory buffers are locked only when needed, for example, when processing a frame (desktop processing) or when a Sparks module is edited. A Sparks plug-in can write to a memory buffer only when its state is locked. When processing, the contents of a buffer are still valid and clean only if the buffer is flagged unlocked and/or clean. If it is flagged dirty, the Autodesk Visual Effects and Finishing product has accessed that memory location, and the buffer contents have therefore been compromised.

Batch Processing in Autodesk Visual Effects and Finishing products

When loaded in the Batch environment of Visual Effects products, a Sparks plug-in has to share its memory resources with other process nodes active in Batch. Sparks memory buffers are

unlocked between processes. If a Sparks memory buffer is used by any other process, its state will become compromised, or dirty. If a Sparks memory buffer is flagged clean, the contents of the buffer are still valid.



Working with Memory Buffers

The following are important factors to consider when working with memory buffers:

- A fixed temporary memory buffer is assumed; however, you must use non-fixed temporary memory buffers if the input or output resolution of your Sparks plug-in could change. You can assume that the validity of pointers or memory buffers will not persist if the input or output resolution changes and that corruption problems may ensue. See "sparkMemRegisterBuffer( )" on page 35.
- If you decide to use the new memory buffer interface, you will benefit from a more flexible interface and your Sparks plug-in will be supported in the Batch processing module of Visual Effects products (the old memory interface does not support Batch processing).
- With the new memory interface, the Sparks plug-in interface calling sequence has been changed: buffer checking done in **`SparkInitialise`** or **`SparkClips`** is no longer valid. Buffer allocation is now checked on-the-fly.
- Memory is only allocated when a memory buffer is flagged `MEMBUF_LOCKED`.

WARNING: A registered buffer is not a locked buffer (that is, you cannot access the Sparks memory unless the buffer is `MEMBUF_LOCKED`).

- Be careful when using **`sparkError( )`** within **`SparkInitialise( )`**. The **`sparkError( )`** function unlinks your Sparks DSO and calls **`SparkUnitialise( )`**.

WARNING: Do not delete any uninitialised memory!

The following is an example of how to use the Sparks memory interface.

```
static int RESULT_ID = 1;
static int CLIP1_ID;
static int AUX1_ID;
static int AUX2_ID;
static int AUX3_ID;
static SparkMemBufStruct SparkResult;
static SparkMemBufStruct SparkClip1;
static SparkMemBufStruct SparkAux1;
static SparkMemBufStruct SparkAux2;
static SparkMemBufStruct SparkAux3;
int SparkClips( void )
{
  return ( 1 );
}
void SparkMemoryTempBuffers( void )
{
  AUX1_ID = sparkMemRegisterBuffer();
  AUX2_ID = sparkMemRegisterBufferSize(
  sizeof(unsigned long)*400 );
  AUX3_ID = sparkMemRegisterBufferFmt(
  200, 100, SPARK_FMT_MONO );
}
unsigned int SparkInitialise( SparkInfoStruct SparkInfo )
{
  int bufCount;
  int i;
   if ( sparkAPIVersionInfo() < 2.5 )
   {
       /* clean here before bailing out! */
```

```
        sparkError("The current IFFFSE version
        "doesn't support this SPARK!");
    }
    /* etc... */
  return ( SPARK_MODULE );
}
unsigned long *SparkProcess( SparkInfoStruct SparkInfo )
{
  int bufCount, i;
  if ( sparkMemGetBuffer(RESULT_ID, &SparkResult) == 0 )
    return NULL;
  if ( sparkMemGetBuffer(CLIP1_ID, &SparkAux1) == 0)
    return NULL;
  if ( sparkMemGetBuffer(AUX1_ID, &SparkAux1) == 0 )
    return NULL;
  if ( sparkMemGetBuffer(AUX2_ID, &SparkAux2) == 0 )
    return NULL;
  if ( sparkMemGetBuffer(AUX3_ID, &SparkAux3) == 0 )
    return NULL;
  if ( !(SparkResult.BufState & MEMBUF_LOCKED )
    ||   !(SparkClip1.BufState  & MEMBUF_LOCKED )
    ||   !(SparkAux1.BufState   & MEMBUF_LOCKED )
    ||   !(SparkAux2.BufState   & MEMBUF_LOCKED )
    ||   !(SparkAux3.BufState   & MEMBUF_LOCKED )
    return NULL;
  bufCount = SparkResult.TotalBuffers;
  /* ...etc. */
  return (SparkResult.Buffer);
}
/* ...etc. */
```

Sparks User Interface

The Sparks user interface employs the same basic control types found in the components of other Autodesk Visual Effects and Finishing products. These control types include an integer numeric display, a floating point numeric display, a text string display, a colour button, a toggle button, a mode display, and a push button.

A specific format is used to name the controls. All control names begin with the string "**Spark**" followed by a string identifying the control type.

Control type string:	Identifies:
Int	An integer numeric display.
Float	A floating point display.
String	A text string display.
Color	A color button.
Boolean	A toggle button.
Popup	A mode display.
Push	A simple push button.

The control name must end in a numeric value that identifies the position of the control on a predefined grid. For example, the control named **SparkFloat0** identifies a floating point numeric display at position 0 on the grid. The control named **SparkPopup21** identifies a mode display at position 21 on the grid.

Text string parameters are wider than the other types of parameters. Typically, they occupy two grid positions, except for those at the desktop level in the extreme right column of the component level.

Control Page Canvas

Sparks plug-ins used in Component mode now have access to the control page canvas by defining an appropriate **SparkCanvasStruct** structure. If a Sparks control is also defined on the control page where a canvas was defined by a Sparks plug-in, the Sparks control will be drawn on top of the canvas. Control page 5 can be used uniquely for a canvas.The following example defines a Sparks canvas on control page 5.

```
SparkCanvasStruct SparkCanvas5 = { HistoDisplayCB,
     HistoInteractCB };
```

These are the related structure declarations added to the *spark.h* header file:

```
typedef struct {
  void (*DisplayCallback) (SparkCanvasInfo);
  int (*InteractCallback)(SparkCanvasInfo Canvas, int
    PointerX, int PointerY, float PointerPressure);
  /* return 1 for canvas display */
} SparkCanvasStruct;


typedef struct {
  int X0; /* Canvas Origin */
  int Y0;
  int Width; /* Canvas Width */
  int Heigth; /* Canvas Height */
}; SparkCanvasInfo;
```

Setup Page Controls

A Sparks plug-in can create up to 22 user interface controls in the Setup menu. You cannot animate these controls. The following example defines an integer numeric display in the Sparks setup menu.

```
SparkIntStruct SparkSetupInt21 = { 5,
  1,
  10,
  1,
  SPARK_FLAG_NO_ANIM,
  "Undo Levels %d",
  NULL } ;
```

Sparks Hot Keys

Sparks plug-ins used in Component mode have standard predefined hot keys. The **F1** through **F4** hot keys can be used to toggle between the input clips and result clip views. The Channel Editor comes with the standard hot keys. Refer to the *Hot Keys Reference Guide* for your product.

Sparks Player

Sparks plug-ins using the new memory model have access to the Player. When a clip is processed, Play and Delete buttons are displayed on the right side on the timebar allowing the user to respectively play or delete the last clip processed.

Desktop and Component Levels

A Sparks plug-in can operate at either or both of two levels: the desktop level and the component level. When a Sparks plug-in is chosen, the Autodesk Visual Effects and Finishing product prompts the user to select all required input clips using the standard clip selection cursors. Enabling the S[ ] box on the Sparks button in the Autodesk Visual Effects and Finishing product lets the user retrieve the same clips used the last time the Sparks plug-in was invoked.

If the Sparks plug-in can operate at both the desktop and the component levels, the E[ ] box in the Autodesk Visual Effects and Finishing product then appears on the Sparks button. If the E[ ] box is selected before selecting the destination reel, the Sparks plug-in runs at the component level; otherwise, it runs at the desktop level. The E[ ] box does not appear if the Sparks plug-in operates only at one of the levels.

The Desktop Level

At the desktop level, the Sparks controls appear in the menu when the user chooses the Sparks plug-in. This is similar to the command menu that appears when the user chooses Dissolve in the Editing menu in Flame.

A Sparks plug-in can have up to six controls at the desktop level. The controls are numbered 0 through 5. No interactive callback functions are associated with the controls at this level. To see the effect of applying a Sparks plug-in to the source clips, the user must set the control values and then process the clips by selecting the destination reel or EditDesk.

The Component Level

At the component level, the Sparks user interface occupies the entire desktop. There is an image window in the work area, control displays and buttons, wide screen viewing tools, and a Channel Editor. This is similar to the interface of the Dissolve Editor, for example, in the Editing menu in Flame. Using a Sparks plug-in at the component level is the best way to debug and fine-tune the Sparks plug-in itself.

The basic controls include timelines, buttons to select the Front, Back, Matte or Result clip for display, buttons to access the Setup, Control or Anim submenus, and an Exit button.

The controls at the component level are numbered from `6` to `121` and may have associated callback functions. You can save your user interface configuration, or setup, and create

animation channels for each user interface variable. An animation channel has the same callback function as that of the corresponding interface control.

Sparks parameters 6 to 34 appear on the first control page; parameters 35 to 63 appear on the second control page; parameters 64 to 92 appear on the third control page; and parameters 93 to 121 appear on the fourth control page.

The user interface buttons on control pages two to four only appear for Sparks plug-ins that define parameters on these pages.

Sample User Interface Control

The following example defines an integer numeric display, or slider, in a Sparks menu.

```
SparkIntStruct SparkInt6 = { 0,
  0,
  100,
  1,
  SPARK_FLAG_X | SPARK_FLAG_NO_ANIM,
  "Slider Value %d",
  NULL } ;
```

In this declaration, the integer numeric display has the following properties:

- The initial value is 0.
- The minimum value is 0.
- The maximum value is 100.
- The increment value is 1. This is used to modify the current value when the pointer device is dragged over the display.
- The flags tell the Autodesk Visual Effects and Finishing product that this parameter is an X coordinate that should be rescaled if the Sparks plug-in is applied to a frame of a different size, and that this parameter should not appear as an entry in the Channel Editor. Other possibilities are SPARK_FLAG_Y (to identify a parameter as a Y coordinate) and SPARK_FLAG_NO_INPUT (to tag a display as an output-only field).
- The text string defines how the integer value appears in the numeric display.
- The Value field corresponds to the integer value you enter.
- The NULL parameter indicates that there is no callback function associated with the numeric display.

- The value of 6 in the control name tells the Autodesk Visual Effects and Finishing product that this control appears at the component level of the Sparks plug-in, on the first control page.

Most Sparks plug-ins are designed to operate at both the desktop and component levels. In order to have the same controls available at both levels, you must declare each desktop level control at the component level as well. For example, the control **`SparkInt0`** operating at the desktop level may have an almost identical control named **`SparkInt6`** operating at the component level.

Other Sparks parameter types have similar fields. Refer to the *spark.h* header file for a complete list of the fields available for each parameter type.

For controls that operate at the two different levels, you may want to include a callback in the declaration of the component level control. A callback updates the image each time you modify the control value (either drag-and-drop or continuous updating). To include a callback in the previous example, replace the `NULL` parameter with the name of a function that is called by the Autodesk Visual Effects and Finishing product each time you modify the control value.

NOTE: The interfaces at the two levels are completely independent; you must declare the same controls at each level.

The Channel Editor

The Sparks user interface includes the standard Channel Editor used by the modules in Autodesk Visual Effects and Finishing products. The Channel Editor is only available at the component level. By default, all Sparks numerical controls (**`int`** and **`float`** types) appear in the Channel Editor unless their Flags field includes the `SPARK_FLAG_NO_ANIM` value.

You can access the Channel Editor with the Animation button available at the component level. When it appears, it replaces the Sparks control page. The standard Channel Editor controls to add, move, and delete keyframes are available. There is also an AutoKey feature that automatically generates a keyframe when a Sparks control is updated, either through the control page or through a call to the **`SparksControlUpdate`** utility function. You can toggle the AutoKey on and off from the Set Up, control, or Channel Editor page.

A control is animated by the Channel Editor if it has one or more keyframes. Otherwise, the control value is taken from the control page.

Adding Controls to the Channel Editor

The Sparks utility functions let you create Channel Editor controls that are not attached to user interface numerical controls. You define these controls using the same structures as user interface controls; the controls can be numbered from `0` to `121`. You can set and retrieve keyframes using Sparks utility functions.

NOTE: The Channel Editor supports a maximum of 244 controls (folders and controls are considered to be the same; that is, you handle both in the same way since Autodesk Visual Effects and Finishing products do not make a distinction).

The following example defines a floating numeric control in the Channel Editor.

```
SparkFloatStruct SparkUserFloat10 = { 0,
  0,
  50,
  0.5,
  SPARK_FLAG_NONE,
  "Amplitude: %d",
  NULL };
```

Controls associated with the Channel Editor are identified by the Sparks User prefix. These controls appear in the Channel Editor only. Unlike regular controls, they are not tied to user interface controls. The flag and callback fields have no effect with Channel Editor controls.

Customizing the Channel Editor Layout

Sparks utility functions let you create channel hierarchies in the Channel Editor. See "Channel Editor Functions" on page 36. You can create up to four levels in the Channel Editor. You do this by adding folders and inserting controls into them. Any controls that you have not added to a folder are appended to the end of the list on the first level.

Using Channel Editor Folders

Note the following when creating Channel Editor folders:

- You activate levels 2, 3, and 4 of the Channel Editor by creating folders.
- You create folders in a parent-child fashion. For example, to create a folder on the second level, a folder must already exist on the first level.
- You must first activate a level before adding a Sparks user interface control or Channel Editor control to it.

- You can create a maximum of four levels in the Channel Editor. Therefore, you cannot add folders to level four.
- Autodesk Visual Effects and Finishing products keep track of the last item on each level. For example, if you create a folder on level one and insert controls into it, and then create another folder on level one, all controls you then insert are added to the newly created folder. This is shown in the following example.

```
void SparkChannelEditor ( void )
{
  sparkCeAddFolder (SPARK_CE_LEVEL1, "offset");
  /* ...creates a folder on level 1 and activates level 2 */
  sparkCeAddControl (SPARK_CE_LEVEL2,
    SPARK_CE_CONTROL, 121);
  /* ...adds the Spark Channel Editor control
  number 121 on level 2 in the offset folder */
  sparkCeAddFolder (SPARK_CE_LEVEL1, "shift");
  /* ...creates a new folder on level 1 */
  sparkCeAddControl (SPARK_CE_LEVEL2,
  SPARK_UI_CONTROL, 121);
  /* ...adds the sparks user interface numerical
    control number 121 on level 2 in the shift folder */
}
```

NOTE: Your UNIX terminal displays all errors that occur when using the **`sparkCeAddFolder`** and the **`sparkCeAddControl`** functions.

Sparks Setup Management

The Sparks Setup menu is also available at the component level from the Sparks user interface. Sparks setups (including all current control values and Channel Editor curves) can be saved, loaded, or deleted.

NOTE: Attempts to load setups containing controls that no longer exist in a Sparks plug-in will fail. When modifying an existing Sparks plug-in, care should be taken to keep it compatible with earlier versions.

3 Sparks Utility Library

Summary

About Sparks Plug-ins

The Sparks development environment includes a number of utility functions that simplify image processing and extend the Sparks user interface. Together, the utility functions form the Sparks utility library. All these functions use the prefix "**`spark`**" (all lowercase) to differentiate them from the "**`Spark`**"-prefixed functions called by Autodesk Visual Effects and Finishing products. The following sections describe the currently available utility functions.

User Interface Functions

Use the user interface functions to show, hide, or control different user interface elements such as the Channel Editor, cursors, and the file browser.

sparkMessage( )

```
void sparkMessage( char *MessageString )
```

The **sparkMessage** function displays the text message supplied by **MessageString** in the desktop message window.

sparkCursorBusy( )

```
void sparkCursorBusy( void )
```

The **sparkCursorBusy** function can be used to change the cursor type while a clip is being processed. This function is called from other user interface callback functions only.

sparkViewingCursor( )

```
void sparkViewingCursor( int CursorIndex )
```

The **sparkViewingCursor** function can be used to define which of the standard cursors in Autodesk Visual Effects and Finishing products should appear when the pointer device is over the image window. All currently available cursors for Autodesk Visual Effects and Finishing products are described in the *spark.h* header file.

NOTE: This list or the appearance of the cursors is subject to change in future releases.

sparkControlUpdate( )

```
void sparkControlUpdate( int ControlNumber )
```

The **sparkControlUpdate** function forces Autodesk Visual Effects and Finishing products to redraw a specific user interface control. This function is required; changing the value associated with a user interface control within the environment of Autodesk Visual Effects and Finishing products does not ensure that the control will be redrawn. The **sparkControlUpdate** function acts as a type of callback within a callback, ensuring consistent updating of user interface controls. The **ControlNumber** parameter is the number used to position the control; this number is associated with the control name. See "Sparks User Interface" on page 19.

sparkReprocess( )

```
void sparkReprocess( void )
```

The **sparkReprocess** function is useful in cases when a Sparks plug-in needs to reprocess the current output frame after a user interface update. Typically, you use this with timewarp Sparks plug-ins where multiple input frames are used to create one output frame. If a user parameter is updated, the **sparkReprocess** function can force a reprocess, causing the Sparks plug-in to receive all input frames again. The Sparks plug-in then generates a new output frame to appear in the image window.

sparkViewingDraw( )

```
void sparkViewingDraw( void )
```

The **sparkViewingDraw** function refreshes the Channel Editor and image window. It also generates a call to **SparkOverlay**, when defined.

sparkMessageConfirm( )

```
int sparkMessageConfirm( char *MessageString )
```

The **sparkMessageConfirm** function displays the text message supplied by **MessageString** in the desktop message window and displays a Confirm button. This function returns `1` when the Confirm button is pressed.

sparkClipControlTitle( )

```
void sparkClipControlTitle( SparkClipSelect Clip, char
*Title )
```

The **sparkClipControlTitle** function lets a Sparks plug-in change default clip control titles. The clip (specified by an enumerator value defined in the *spark.h* header file) title is replaced with the new title.

sparkPointerRead( )

```
int sparkPointerRead( void )
```

The **sparkPointerRead** function returns the pointer device pressure value.

sparkPointerWaitOff( )

```
void sparkPointerWaitOff( void )
```

The **sparkPointerWaitOff** function waits and returns when `0` pressure is applied to the pointer device.

sparkPointerWaitOn( )

```
void sparkPointerWaitOn( void )
```

The **sparkPointerWaitOn** function waits and returns when pressure is applied to the pointer device.

sparkQueryKeyMap( )

```
int sparkQueryKeyMap( long Device )
```

The **sparkQueryKeyMap** function returns the logical state of a device you want to test. The Device parameter corresponds to the old device codes extracted from *gl.h* and *device.h* (on IrisGL systems).

sparkError( )

```
void sparkError( char *ErrorString )
```

The **sparkError** function displays the error message supplied by **ErrorString** in the desktop message window and aborts the current Sparks plug-in.

sparkMpFork( )

```
void sparkMpFork( void (*Function()), int NumArgs, ... )
```

The **sparkMpFork** function is used for multiprocessing a user-defined function. The **Function** argument is the name of the function to be multiprocessed. The **NumArgs** argument specifies the number of arguments required by the **Function**. The other arguments of **sparkMpFork** are the arguments of the function to be multiprocessed; you can use a maximum of six arguments.

It is your responsibility to ensure that the task is correctly allocated to the different CPUs. The total number of processors available for the task is contained in the **NumProcessors** field of the structure **SparkInfoStruct**, and can be extracted using the **SparkInitialise** function.

sparkMpInfo( )

```
void sparkMpInfo( unsigned long *offset, unsigned long
*pixels )
```

The **sparkMpInfo** function calculates an offset in the image buffer to be processed as well as the total number of pixels to be processed. The calculation is based on the number of available processors and on the processor currently being used by the Sparks plug-in.

You should call the **sparkMpInfo** function from the function that is passed as the first argument of the **sparkMpFork** function.

sparkMpForkPixels( )

```
void sparkMpForkPixels( void ( *Function ) ( ), ulong Pixels,
int NumArgs,... )
```

The **sparkMpForkPixels** function is a subset of the **sparkMpFork** function. The **Pixels** argument lets you determine how to divide pixels between processes.

sparkMpIsMainTask( )

```
void sparkMpIsMainTask( void )
```

The **sparkMpIsMainTask** function will return TRUE when called from the main task and return FALSE otherwise. Tasks are usually created via calls to **sparkMpFork**, **sparkMpForkPixels**, or **sparkMpCreateTask**.

sparkProcessTruncate( )

```
void sparkProcessTruncate( void )
```

You can use the **sparkProcessTruncate** function within the **SparkProcess** function to truncate a currently processing clip.

sparkDisableParameter( )

```
void sparkDisableParameter(int Type, int ControlNo)
```

The **sparkDisableParameter** function can be used to hide the UI parameter specified by **ControlNo** in the control page and the Channel Editor.

sparkEnableParameter( )

```
void sparkEnableParameter(int Type, int ControlNo)
```

The **sparkEnableParameter** function can be used to show the UI parameter specified by **ControlNo** in the control page and the Channel Editor.

sparkControlTitle( )

```
void sparkControlTitle(SparkControlSelect Control, char
*Title)
```

The **sparkControlTitle** function can be used to rename the control page parameter identified by **Control**.

sparkResultClipName( )

```
void sparkResultClipName(char *NewName)
```

The **sparkResultClipName** function can be used to set the name of the clip processed by a Sparks plug-in. By default, the Sparks plug-in will use the Sparks plug-in name as the clip name.

sparkPointerInfo( )

```
float sparkPointerInfo(int *PX, int *PY)
```

The **sparkPointerInfo( )** function can be used to get the current pointer device position. This function returns the current pointer pressure.

sparkGetViewerRatio( )

```
float sparkGetViewerRatio(void)
```

The **sparkGetViewerRatio( )** returns the current ratio value applied in the image window.

sparkFrameRate( )

```
double sparkFrameRate( )
```

The **sparkFrameRate( )** function returns the current video frame rate in frames per second.

sparkSetupControlUpdate ( )

```
void sparkSetupControlUpdate( int ControlNo )
```

The **sparkSetupControlUpdate** function forces the application to redraw a specific user interface setup control.

sparkFileBrowserDisplayLoad( )

```
void sparkFileBrowserDisplayLoad( char *Path, char
*Extension, void (*Callback) (char *FileName) )
```

The **sparkFileBrowserDisplayLoad** function displays the file browser and sets the current search path to Path. Extension is used to filter the file. **Callback** will be called on the user-selected file. The selected file is passed as an argument to the callback.

sparkFileBrowserDisplayLoadSequence( )

```
void sparkFileBrowserDisplayLoadSequence( char *Path, char
*Extension, void (*Callback) (char *FileName) )
```

The **sparkFileBrowserDisplayLoadSequence** function displays the file browser and sets the current search path to Path, where the user can select multiple files. Extension is used to filter the file. The selected files are passed as an argument to the **Callback** one by one; the **Callback** is called as many times as there are selected files.

sparkFileBrowserDisplaySave( )

```
void sparkFileBrowserDisplaySave( char *Path, char
*Extension, void (*Callback) (char *FileName) )
```

The **sparkFileBrowserDisplaySave** function displays the file browser and sets the current search path to Path. Extension is used to filter the file. **Callback** will be called on the user-selected file. The selected file is passed as an argument to the callback.

sparkFileCheckOverwrite( )

```
int sparkFileCheckOverwrite( const char *Name )
```

The **sparkFileCheckOverwrite** function can be used within the **sparkFileBrowserDisplayLoad** or **sparkFileBrowserDisplaySave** callbacks to prompt the user with the standard confirm message.

sparkFileHasExtension( )

```
int sparkFileHasExtension(const char *File, const char
*Extension)
```

The **sparkFileHasExtension** function returns `1` if File has an extension equal to Extension. This function is case sensitive. It returns `0` otherwise.

System Functions

Use the system functions to bypass the **sprocs**, **pcreate**, **fork**, and **exec** calls that are no longer available.

sparkSystemSh( )

```
int sparkSystemSh( int wait, const char *cmd )
```

The **sparksSystemSh** function is a wrapper to **pcreatel()**. If the **wait** argument is TRUE, the function will wait for the child process to complete before returning. The **cmd** argument will be passed to a subshell (*/bin/sh*), which will perform the path search—with wildcard expansion if required—as well as break up the command line into separate arguments to be passed to the process that has started.

sparkSystemNoSh( )

```
int sparkSystemNoSh( int wait, const char *path, const char
**argv )
```

The **sparksSystemNoSh** function is a wrapper to **pcreatev()**. If the **wait** argument is TRUE, the function will wait for the child process to complete before returning. The **path** argument points to the executable. The **argv** argument is a pointer to an array of strings representing the parameters to the subprocess. Usually **argv[0]** is the path to the executable; no path searching is done and no wildcard expansion is performed.

NOTE: The last entry in **argv[]** should be a zero to indicate its end.

sparkWaitpid( )

```
int sparkWaitpid( pid_t pid, int *statptr, int option )
```

The **sparkWaitpid** function is a wrapper to **waitpid()**.

The **pid** argument is the value returned from the **sparksSystemSh** or **sparksSystemNoSh** functions, if used with a FALSE **wait** argument.

If the **statptr** argument is non-zero, 16 bits of information called "status" are stored in the low-order 16 bits of the memory location that the **statptr** argument points to. See the **waitpid** man page for details.

NOTE: The options argument is constructed from the bitwise inclusive "OR" of zero or more of the "WNOHANG" and "WUNTRACED" flags.

WARNING: Using **sparkWaitpid** in blocking mode will block any other thread that tries to perform a **waitpid** operation. It is therefore recommended to always use **sparkWaitpid** in non-blocking mode by enabling the WNOHANG flag.

Memory Functions

Use the memory functions to register buffers, set information based on the type of buffer, and estimate the number of buffers that the system can accommodate.

sparkMemRegisterBuffer( )

```
int sparkMemRegisterBuffer( void )
```

The **sparkMemRegisterBuffer** function registers Autodesk Visual Effects and Finishing product-sized buffers. The buffer definition corresponds to one of the structures defined in the *spark.h* header file. The size of registered buffers is determined by the current resolution and current bit depth of the Sparks plug-in. If the output resolution or bit depth of the Sparks plug-in changes, existing memory buffers will be deleted and new ones registered. The function returns the buffer ID.

NOTE: Because the input and output resolution of the Sparks plug-in can change, memory buffers are not valid across resolution changes and cached pointers cannot be relied on to point to valid data.

WARNING: Image buffers should be requested through the memory management system of Autodesk Visual Effects and Finishing products using API calls. As of Inferno 5.0, Flame 8.0, Flint 8.0, Smoke 5.2, the use of dynamic memory allocation of image buffers through **malloc**, **new,** or other similar functions is strongly discouraged. If the memory management of the Autodesk Visual Effects and Finishing product cannot satisfy a request for memory, calls to **malloc**, **new**, or other similar functions are unlikely to succeed. Immediate and violent program termination may ensue; however, you can use dynamic memory allocation for runtime objects and data structures.

sparkMemRegisterBufferSize( )

```
int sparkMemRegisterBufferSize( unsigned long size )
```

The **sparkMemRegisterBufferSize** function registers a buffer of the specified size. The function returns the buffer ID.

sparkMemRegisterBufferFmt( )

```
int sparkMemRegisterBufferFmt( int width, int
height,SparkBufferFmt )
```

The **sparkMemRegisterBufferFmt** function registers a buffer of the specified format. The function returns the buffer ID.

sparkMemGetBuffer( )

```
int sparkMemGetBuffer( int id, SparkMemBufStruct *memBufInfo
)
```

When passed the ID of a registered buffer and a **SparkMemBufStruct**, this function will set all the information relative to this buffer. This function returns `1` if successful.

sparkMemGetFreeMemory( )

```
int sparkMemGetFreeMemory( )
```

The **sparkaMemGetFreeMemory** function returns an estimate of the maximum number of Autodesk Visual Effects and Finishing product-sized memory buffers that can be locked at the same time by a Sparks plug-in.

NOTE: Depending on which type of **sparkMemRegisterBuffer** is used, Autodesk Visual Effects and Finishing products will provide more or less information in the **SparkMemBufStruct** with a buffer. The **SparkBufferFmt** and **SparkMemBufStruct** fields used by the Memory functions are defined in the *spark.h* header file.

Channel Editor Functions

Use the Channel Editor functions to control keyframes, folders, and the names of controls in the Channel Editor.

sparkSetCurveKey( )

```
void sparkSetCurveKey( int Type, int ControlNumber, int
FrameNumber, float Value)
```

You can use the **sparkSetCurveKey** function to set keyframe values from a Sparks plug-in. These keyframe values are assigned to the control specified by the **Type** and **ControlNumber** parameters.

sparkSetCurveKeyf( )

```
void sparkSetCurveKeyf( int Type, int ControlNo, float Frame,
float Value)
```

The **sparkSetCurveKeyf** function can be used to set keyframes between frames. The keyframe values are assigned to the control specified by the **Type** and **ControlNo** parameters. See "sparkSetCurveKeyf( )" on page 51.

sparkGetCurveValuef( )

```
float sparkGetCurveValuef( int Type, int ControlNumber,
float FrameNumber )
```

The **sparkGetCurveValuef** function returns the keyframe value of the control specified by the **Type** and **ControlNumber** parameters.

sparkGetCurveValue( )

```
float sparkGetCurveValue( int Type, int ControlNumber, int
FrameNumber )
```

The **sparkGetCurveValue** function is the same as the **sparkGetCurveValuef** function except that it takes an **int** value for **FrameNumber** rather than a float value.

sparkIsAutoKeyOn ( )

```
void sparkIsAutoKeyOn (int)
```

The **sparkIsAutoKeyOn** function returns `1` if the AutoKey function is enabled and `0` if the AutoKey function is disabled.

sparkCeAddFolder( )

```
void sparkCeAddFolder( int Level, char *Title )
```

You use the **sparkCeAddFolder** function to create folders in the Channel Editor. Adding a folder creates a new sub-level in the Channel Editor into which you can add Channel Editor or Sparks user interface controls. The level parameter indicates to which active level the folder should be added. The title parameter is used as a folder name.

sparkCeAddControl( )

```
void sparkCeAddControl( int Level, int Type, int
ControlNumber )
```

By default all numerical Sparks user interface controls appear in the Channel Editor on the first level. You use the **sparkCeAddControl** function to place these controls in the folder created by the **sparkCeAddFolder** function. The **Type** and **ControlNumber** parameters specify which item is affected.

sparkSetControlName( )

```
int sparkSetControlName( int Type, int ControlNumber, char
*NewName )
```

The **sparkSetControlName** function sets the name of a control appearing in the Channel Editor to **NewName**. This function works only if the Sparks plug-in is not in Channel Editor mode and returns 1 when the renaming occurs. The **Type** and **ControlNumber** parameters specify which item is affected.

NOTE: The **Type** and **Level** fields used by the Channel Editor functions are enumerators defined in the *spark.h* header file.

sparkChRemoveKey( )

```
void sparkChRemoveKey(int ControlType, int ControlNo, float
Frame)
```

The **sparkChRemoveKey( )** function removes the key specified by **Frame** of the given control and control type.

sparkChClear( )

```
void sparkChClear(int ControlType, int ControlNo)
```

The **sparkChClear( )** function removes all keyframes of the given control and control type.

sparkChCopy( )

```
void sparkChCopy(int SrcControlType, int SrcControlNo, int
DstControlType, int DstControlNo)
```

The **sparkChCopy( )** function copies the keyframes of the **SrcControlNo** to the **DstControlNo**.

sparkChPreComputeValues ( )

```
int sparkChPreComputeValues(int ControlType, int ControlNo,
float FrameNo)
```

The **sparkChPreComputeValues( )** function computes the values for the channel identified by **ControlType** and **ControlNo** between **FrameStart** and **FrameEnd**. The **sparkChPreComputeValues( )** function returns SPARK_FAILURE in the following situations:

- If **ControlNo** has no associated channel.
- If **ControlNo** is outside the range.
- If the Channel Editor has not been initialised when the call is made to **sparkChPreComputeValues( )**.

sparkChPreComputedValues( )

```
float sparkChPreComputedValues(int ControlType, int
ControlNo, float FrameNo)
```

The **sparkChPreComputedValues( )** function retrieves a previously computed channel value. **sparkChPreComputedValues( )** returns a value of `0.f` when an error occurs.

sparkChReadRawKeys( )

```
int sparkChReadRawKeys(const char *FileName, int
ControlType, int ControlNo)
```

The **sparkChReadRawKeys( )** function can be used to read the keyframes of the given **FileName** ASCII file. The current channel data will be overwritten by the new data. The function returns `1` if the file was successfully read.

Environment Functions

Use the environment functions to set the parameters of the graphics monitor, determine the Sparks API version, the directory of the Sparks plug-in, or to determine which module will call the Sparks plug-in.

sparkGraphSetup( )

```
void sparkGraphSetup( SparkGraphInfoStruct *SparkGraphInfo )
```

The **sparkGraphSetup** function returns a structure containing the resolution and pixel aspect ratio of the viewing area of the graphics monitor.

sparkAPIVersionInfo( )

```
float sparkAPIVersionInfo( void )
```

The **sparkAPIVersionInfo** function returns the version number of the system Sparks API.

sparkProgramGetName( )

```
const char* sparkProgramGetName( void )
```

The **sparkProgramGetName** function returns a character string corresponding to the name of the Autodesk Visual Effects and Finishing product in which the Sparks plug-in is currently loaded.

sparkGetInfo( )

```
void sparkGetInfo( SparkInfoStruct *SparkInfo)
```

The **sparkGetInfo** function can be used to obtain information about the resolution of the current Sparks plug-in.

sparkWorkingDir( )

```
void sparkWorkingDir( char *Dir )
```

The **sparkWorkingDir** function sets the **Dir** string to the working directory of a Sparks plug-in.

sparkCallingEnv( )

```
int sparkCallingEnv( void )
```

The **sparkCallingEnv** function returns an enumerator value corresponding to the module calling the Sparks plug-in. These values are defined in the *spark.h* header file.

Image Access Functions

Use this function to return the value of the requested frame.

sparkGetFrame( )

```
void sparkGetFrame( SparkClipSelect Clip, int FrameIndex,
unsigned long *Destination )
```

From the desktop, the **sparkGetFrame** function returns the requested frame of the requested clip into the destination buffer. In Batch, **sparkGetFrame** updates the input node corresponding to the requested frame using the destination buffer for the resulting image. The destination buffer must be a registered result or a temporary buffer, in RGB format, previously retrieved with **sparkGetBuffer**. The index of the first frame in a clip is `0`.

Image Colour Space Conversion Functions

Use the colour space conversion functions to determine which colour model to use for the image. You can also convert images between different colour models or create a monochrome or negative image.

sparkMonochrome( )

```
void sparkMonochrome( unsigned long *Source, unsigned long
*Destination)
```

The **sparkMonochrome** function fills the destination buffer with a monochrome version of the image in the source buffer.

sparkNegative( )

```
void sparkNegative( unsigned long *Source, unsigned long
*Destination )
```

The **sparkNegative** function fills the destination buffer with a negative (inverted) version of the image in the source buffer.

sparkToYUV( )

```
void sparkToYUV(unsigned long *Source, unsigned long
*Destination)
```

The **sparkToYUV** function fills the destination buffer with a YUV (Chrominance/ Luminance) version of the image in the source buffer. The channels are mapped as follows:

• G is mapped to Y (luminance).

• B is mapped to U (B chrominance).

• R is mapped to V (R chrominance).

Regardless of the colour space used, all channels are mapped to the range of unsigned 8-bit integer values.

sparkToHLS( )

```
void sparkToHLS(unsigned long *Source, unsigned long
*Destination)
```

The **sparkToHLS** function fills the destination buffer with an HLS (Hue, Luminance, Saturation) version of the image in the source buffer. The channels are mapped as follows:

• R is mapped to H (Hue).

• G is mapped to L (Luminance).

• B is mapped to S (Saturation).

Regardless of the colour space used, all channels are mapped to the range of unsigned 8-bit integer values.

sparkFromYUV( )

```
void sparkFromYUV( unsigned long *Source, unsigned long
*Destination )
```

The **sparkFromYUV** function fills the destination buffer with an RGB version of the image in the source buffer. The channels are mapped as follows:

• V (R chrominance) is mapped to R.

• Y (luminance) is mapped to G.

• U (B chrominance) is mapped to B.

sparkFromHLS( )

```
void sparkFromHLS( unsigned long *Source,unsigned long
*Destination )
```

The **sparkFromHLS** function fills the destination buffer with an RGB version of the image in the source buffer. The channels are mapped as follows:

• H (Hue) is mapped to R.

• L (Luminance) is mapped to G.

• S (Saturation) is mapped to B.

sparkRGBtoYUV( )

```
void sparkRGBtoYUV( int R, int G, int B, int *Y, int *U, int
*V)
```

The **sparkRGBtoYUV** function produces the Y, U, and V output values that correspond to the R, G, and B input values.

sparkRGBtoHLS( )

```
void sparkRGBtoHLS( int R, int G, int B, int *H, int *L, int
*S)
```

The **sparkRGBtoHLS** function produces the H, L, and S output values that correspond to the R, G, and B input values.

sparkYUVtoRGB( )

```
void sparkYUVtoRGB( int Y, int U, int V, int *R, int *G, int
*B )
```

The **sparkYUVtoRGB** function produces the R, G, and B output values that correspond to the Y, U, and V input values.

sparkHLStoRGB( )

```
void sparkHLStoRGB( int H, int L, int S, int *R, int *G, int
*B )
```

The **sparkHLStoRGB** function produces the R, G, and B output values that correspond to the H, L and S input values.

Image-Processing Functions

Create a blurred or a composite version of the clip with the image-processing functions.

sparkBlur( )

```
void sparkBlur( unsigned long *Source, unsigned long
*Destination, int x, int y, int Channels )
```

The **sparkBlur** function fills the destination buffer with a blurred version of the image in the source buffer. The x and y parameters define the severity of the blur in the horizontal and vertical directions, respectively. A high value for either x or y produces a more severe blur. For a uniform blur, make x equal to y. The Channels parameter determines which channels are affected by the blur.

sparkComposite( )

```
void sparkComposite( unsigned long *Front, unsigned long
*Back, unsigned long *Matte, unsigned long *Destination )
```

The **sparkComposite** function produces a composite image using the input Front, Back, and Matte image buffers. The Matte image is assumed to be black and white, with identical R, G, and B values for each pixel. It acts as the alpha channel for the Front image. For each pixel in the Matte image, if the pixel has a non-zero RGB value, then a pixel from the Front image is weighted according to the Matte RGB value and appears in the destination image. If the pixel in the Matte image has a zero RGB value (a black pixel), then a pixel from the Back image appears in the destination image.

Image Buffer Manipulation Functions

Copy the entire contents or the contents of a particular channel from the source to destination buffer using the image buffer manipulation functions. You can also change the size of the destination image from that of the source image.

sparkCopyBuffer( )

```
void sparkCopyBuffer( unsigned long *Source, unsigned long
*Destination)
```

The **sparkCopyBuffer** function copies the contents of the source buffer into the destination buffer.

sparkCopyChannel( )

```
void sparkCopyChannel( unsigned long *Source, int
SourceChannel, unsigned long *Destination, int
DestinationChannel )
```

The **sparkCopyChannel** function copies the specified `SourceChannel` of the source buffer into the `DestinationChannel` of the destination buffer.

sparkResizeBuffer( )

```
void sparkResizeBuffer( unsigned long *Source, int
SourceSize, unsigned long *Destination, int DestinationSize
)
```

The **sparkResizeBuffer** function may be used to convert the format of the image buffers. Use the enumerator values located in the *spark.h* header file to specify the source and destination sizes.

File I/O Support Functions

Determine ownership and access privileges of a file using the file I/O support functions. Also set whether a setup file is saved or loaded.

sparkSetPermissions( )

```
void sparkSetPermissions( char *filename )
```

The **sparkSetPermissions** function changes the ownership and access permissions of the given file to the standard of the Autodesk Visual Effects and Finishing product. This function should be used on every permanent file created by a Sparks plug-in to ensure that all users of Autodesk Visual Effects and Finishing products on a given system are able to access the file.

sparkSaveSetup( )

```
void sparkSaveSetup( char *filename )
```

The **sparkSaveSetup** function saves the current Sparks setup (including all current user interface values and Channel Editor curves) into a file with the given name. The user can load a saved setup with the **sparkLoadSetup** function. The setup files of Autodesk Visual Effects

and Finishing products are stored in the */usr/discreet/project/<project name>/sparks* of the current project, which is defined on the Sparks line of the configuration file of the Autodesk Visual Effects and Finishing products and is the name of the Sparks plug-in.

sparkLoadSetup( )

```
void sparkLoadSetup( char *filename )
```

The **sparkLoadSetup** function loads a setup previously saved with the **sparkSaveSetup** function.

Scan Mode Identification Function

Detemine the scan mode of a Sparks setup.

sparkGetScanFormat( )

```
sparkGetScanFormat( )
```

The **sparkGetScanFormat** function returns the scan mode of the current Sparks setup according to the following enumerated type definitions:

```
typedef enum {
   SPARK_SCAN_FORMAT_UNDEFINED = -1,
   SPARK_SCAN_FORMAT_FIELD_1 = 0,
   SPARK_SCAN_FORMAT_FIELD_2 = 1,
   SPARK_SCAN_FORMAT_PROGRESSIVE = 2
} SparkScanFormat;
```

Process Management Functions

Set up and control multi-process tasks.

sparkMpAllocateTaskHandle( )

```
SparkTaskHandle_t* sparkMpAllocateTaskHandle( )
```

The **sparkMpAllocateTaskHandle** function allocates a handle to create a multi-process task.

int sparkMpCreateTask( )

```
int sparkMpCreateTask( SparkTaskHandle_t* handle, const
char* name, int cpu, SparkTaskFunc_t* entry, void* arg )
```

The **sparkMpCreateTask** function creates a multi-process task on the cpu supplied. If the cpu is -1, the main thread cpu is used. The function returns 0 on success and -1 on failure.

sparkMpWaitTask( )

```
void sparkMpWaitTask( SparkTaskHandle_t* handle )
```

The **sparkMpWaitTask** function creates a pause in what is being executed until the task related to the handle indicated is completed.

sparkMpFreeTaskHandle( )

```
void sparkMpFreeTaskHandle( SparkTaskHandle_t* handle )
```

The **sparkMpFreeTaskHandle** function frees the handle.

sparkMpGetCpu( )

```
int sparkMpGetCpu( )
```

The **sparkMpGetCpu** function returns which cpu the current thread is running on.

4 Sparks Audio API

Summary

About Sparks Audio API

This chapter describes audio-related functions. All described and referred functions can be found in the *spark.h* header file in the *~/sparks* directory.

In addition to reading this chapter, Sparks developers should examine the **`sparkAudioPreviewGain`** example, as well as the **`sparkAudioAPI`**, **`sparkMono2Stereo`**, **`sparkDisplaySound`** and **`sparkReverse`** sample Sparks source files found in the *~/sparks* directory. **`sparkAudioAPI`** tests one by one the new audio-related functions. **`sparkMono2Stereo`** shows how to manipulate audio tracks block by block. **`sparkDisplaySound`** shows how to relate samples with the current frame within **`SparkProcess( )`**. **`sparkReverse`** reimplements clip reversal with audio support.

Global Audio Parameter Access Functions

The global audio parameter access functions return information about audio samples, and values such as audio block size and space available for audio (in bytes and number of tracks).

sparkSampleFormat( )

`SparkBufferFmt sparkSampleFormat( )` returns the sample format.

Currently, the only possible return value is `SPARK_FMT_AUDIO_INT16`.

sparkSampleWidth( )

The **sparkSampleWidth( )** function returns the sample width in bytes.

Currently, the only possible return value is 2.

sparkAudioBlockSize( )

The **sparkAudioBlockSize( )** function returns the internal audio block size in bytes.

sparkAudioFreeSpace( )

uint64_t sparkAudioFreeSpace( ) returns the free audio space available in bytes.

sparkAudioMaxOutputTracks( )

The **sparkAudioMaxOutputTracks( )** function returns the maximum number of output audio tracks.

sparkFrameRate( )

double sparkFrameRate( ) returns the current video frame rate in frames per second.

sparkSamplingRate( )

double sparkSamplingRate( ) returns the current audio sampling rate in samples per second.

Memory Buffer Management

Memory allocation when working with audio clips differs from when working with video clips. Use the following memory buffer management function when working with audio clips.

typedef enum spark_buffer_fmt { } SparkBufferFmt;

```
typedef enum spark_buffer_fmt { SPARK_FMT_UNKNOWN = 0,
SPARK_FMT_MONO, SPARK_FMT_FLOAT, SPARK_FMT_RGB,
SPARK_FMT_HLS, SPARK_FMT_YUV, SPARK_FMT_RGBA,
SPARK_FMT_HLSA, SPARK_FMT_YUVA, SPARK_FMT_UYVY,
SPARK_FMT_AUDIO_INT16 = 100 } SparkBufferFmt;
```

SPARK_FMT_AUDIO_INT16 defines an audio-specific memory buffer format. Audio samples of this format are 16-bit signed integers.

Memory-aligned audio buffers can be allocated with **sparkMemRegisterBufferFmt( )**. Parameter **width** gives the number of samples per track and parameter **height** gives the number of tracks. Set **width** to a multiple of **sparkAudioBlockSize( ) /**

sparkSampleWidth( ) to optimize buffer copy and write operations. The sample buffers are not interleaved because audio is handled as mono tracks in Autodesk Visual Effects and Finishing products.

Allocated memory-aligned audio buffers can be retrieved with **sparkMemGetBuffer( )**. The following fields in the filled **SparkMemBufStruct** structure can be interpreted as: **BufWidth** is the number of samples in a track; **BufHeight** is the number of tracks; **BufDepth** is the number of bits per sample; **Stride** is the length to the next track in bytes; and **Inc** is the increment in bytes to move to the next sample.

Note that **sparkCopyBuffer**, **sparkResizeBuffer** and **sparkCopyChannel** are image buffer manipulation functions so they should not be used on memory-aligned audio buffers.

SparkClipInfoStruct and SparkTrackInfoStruct

The following two functions are used for carrying information about the content of the video clip and the content of the audio track.

typedef struct spark_clip_info_struct { } SparkClipInfoStruct;

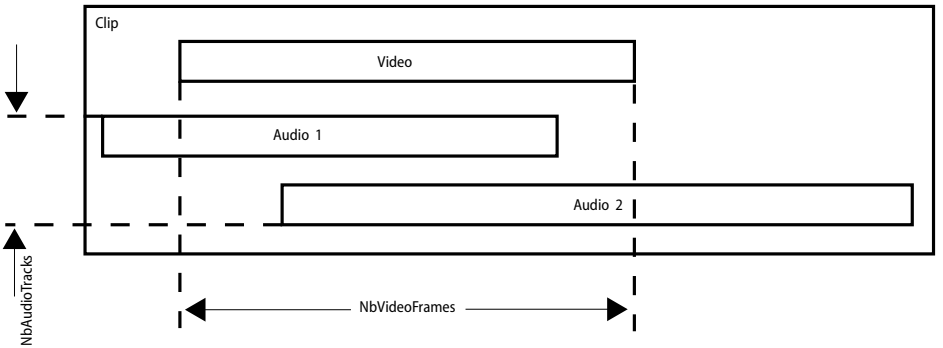
```
typedef struct spark_clip_info_struct { int NbVideoFrames;
int NbAudioTracks; ulong reserved[6]; } SparkClipInfoStruct;
```

typedef struct spark_track_info_struct { } SparkTrackInfoStruct;

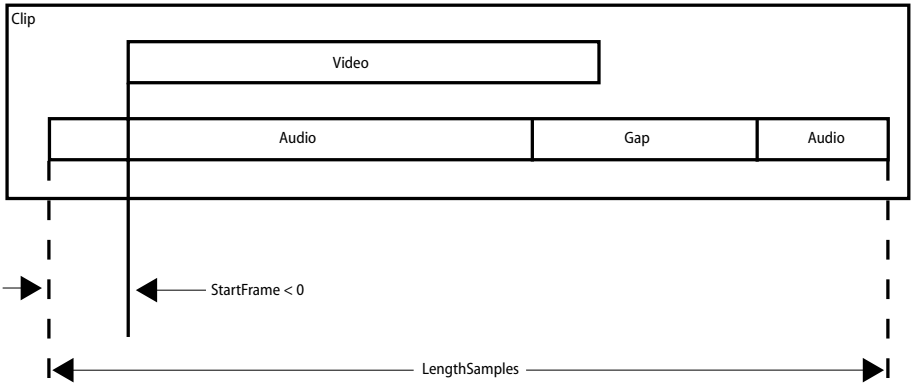
```
typedef struct spark_track_info_struct { float StartFrame
int LengthSamples; ulong reserved[6]; }
SparkTrackInfoStruct;
```

SparkClipInfoStruct carries the information about the content of a clip. **NbVideoFrames** gives the number of video frames in the clip. It is set to 0 if the clip has no

video content. **`NbAudioTracks`** gives the number of audio tracks in the clip. It is set to `0` if the clip has no audio content.



`SparkTrackInfoStruct` carries the information about the content of an audio track. **`StartFrame`** is the audio track's offset in frame units from the first video frame. It is negative if the track starts before the first video frame. **`LengthSamples`** is the number of samples in the track.



Processing and Analyse Functions

The following functions are used to control when processes begin and end.

SparkProcessEnd( )

```
void SparkProcessEnd( SparkInfoStruct SparkInfo )
```

SparkAnalyse( )

```
ulong * SparkAnalyse( SparkInfoStruct SparkInfo )
```

SparkAnalyseStart( )

```
int SparkAnalyseStart( SparkInfoStruct SparkInfo )
```

SparkAnalyseEnd( )

```
void SparkAnalyseEnd( SparkInfoStruct SparkInfo )
```

The audio tracks are carried from source to result unless:

- There has been at least one call to **`sparkNewAudio( )`** between the end of the previous processing loop and the end of the current one.
- Audio output has been disabled with a call to **`sparkAudioOutputEnable( )`**.
- The Sparks plug-in is called from Batch.

The source clip to copy the audio tracks from is selected with the following precedency: front, back, matte. If **`SparkProcess( )`** returns `NULL` when called in the processing loop, the Player displays a black frame and no frame is appended to the output clip. Note that **`SparkProcess( )`** must return a valid pointer to a frame buffer if the Sparks plug-in is called from Batch. **`SparkProcessStart( )`** is no longer mandatory. If not defined, it is assumed to return `1`. **`SparkProcessEnd( )`** is executed once at the end of the processing loop whether it is completed or aborted. This function is not mandatory.

`SparkAnalyse( )` works similarly to **`SparkProcess( )`** but does not generate an output clip. This function is not mandatory. **`SparkAnalyseStart( )`** is used to control the length ratio between the input and the output just as **`SparkProcessStart( )`** does. If not defined, it is assumed to return `1`. **`SparkAnalyseEnd( )`** is executed once at the end of the analysis loop whether it is completed or aborted. This function is not mandatory.

The field **`SparkInfo.TotalFrameNo`** passed to all these functions gives the maximum video length among the input clips. If none of the input clips contains video frames, **`SparkInfo.TotalFrameNo`** is set to `1`.

sparkSetCurveKeyf( )

```
void sparkSetCurveKeyf( int type, int controlNo, float
frameNo, float value)
```

`sparkSetCurveKeyf( )` sets curve keyframes between frames. See "sparkSetCurveKeyf( )" on page 36. To set curve keyframes relative to a sample index, use the following conversion formula:

```
frameNo =  ( audio element start frame ) + ( sample index ) *
           ( sparkFrameRate() / sparkSamplingRate() )
```

Other Audio Functions

The following functions are also used when working with audio. These include functions for filtering clip selections, getting clip information, reading new audio elements for input, allocating new audio elements for output, and working with audio samples.

SparkClipFilter( )

```
int SparkClipFilter (SparkClipSelect Clip,
SparkClipInfoStruct ClipInfo)
```

SparkClipFilter( ) filters each clip selection. If not defined, the default filter condition is to accept a clip with at least one video frame. **Clip** identifies the clip currently filtered and **ClipInfo** contains the clip information. This function is not mandatory. It is disabled if the Sparks plug-in is called from Batch.

Returns SPARK_SUCCESS to accept the current clip selection or SPARK_FAILURE to reject it.

sparkGetClipInfo( )

```
int sparkGetClipInfo( SparkClipSelectClip,
SparkClipInfoStruct * SparkClipInfo )
```

sparkGetAudioTrackInfo( )

```
int sparkGetAudioTrackInfo( SparkClipSelect Clip, int
TrackNo, SparkTrackInfoStruct * SparkTrackInfo )
```

Both functions give access to clip information. These functions can be called on a clip once it is selected or within **SparkClipFilter( )** to get more information on the filtered clip. **Clip** identifies the clip to get the information from. In **sparkGetAudioTrackInfo( )**, **TrackNo**'s allowable values range from 1 to the number of audio tracks in the clip. These functions are disabled if the Sparks plug-in is called from Batch.

Both functions return SPARK_SUCCESS on success or SPARK_FAILURE on failure.

sparkAudioOutputEnable( )

```
void sparkAudioOutputEnable( int Status )
```

By default, tracks are copied from source to output clip if no new audio track is created. If **sparkAudioOutputEnable( )** is called with **Status** set to **FALSE**, no audio data is copied in the output clip whether it comes from a source clip or from created tracks. Call **sparkAudioOutputEnable( )** with **Status** set to **TRUE** to return to the default behaviour. This function has no effect if the Sparks plug-in is called from Batch.

sparkReadAudio( )

```
int sparkReadAudio( SparkClipSelect Clip, int TrackNo, int
StartIndex, int LengthSamples void * Buffer )
```

sparkReadAudio( ) reads audio samples from an input clip audio track. **Clip** identifies the clip to get the samples from. **TrackNo**'s allowable values range from 1 to the number of audio tracks in the clip. **StartIndex** is the index of the first sample to read. **LengthSamples** is the number of samples to read. If the interval requested falls out of the track's limits, missing samples are set to 0. **Buffer** is a pointer to a memory area at least **LengthSamples*sparkSampleWidth( )** bytes long. Data transfer is optimized when **StartIndex** and **LengthSamples** are multiples of **sparkAudioBlockSize( ) / sparkSampleWidth( )** and **Buffer** refers to memory aligned on a page size boundary. This function is disabled if the Sparks plug-in is called from Batch.

sparkReadAudio( ) returns the number of samples read.

sparkNewAudio( )

```
int sparkNewAudio( int TrackNo, float StartFrame, int
LengthSamples int SetToZero )
```

sparkNewAudio( ) allocates and opens a new audio element in the output clip. **TrackNo** is the number of the track where the new audio element is created. **TrackNo** allowable values range from 1 to **sparkAudioMaxOutputTracks( )** return value. **StartFrame** is the new audio element's offset in frame units from the output clip's first video frame. It is negative if the new audio element starts before the first video frame. **LengthSamples** is the number of samples in the new audio element. If **SetToZero** value is **TRUE**, all samples in the new audio element are set to zero. If **sparkNewAudio( )** is called more than once on a track, the previously allocated audio element in this track is closed and copied to the output clip.

At the end of the process loop, after **SparkProcessEnd( )** is called if defined, all the remaining opened audio elements are closed and copied to the output clip. If two audio elements created in the same output track overlap, a crossfade is inserted between them. This function is disabled if the Sparks plug-in is called from Batch.

sparkNewAudio( ) returns the number of samples allocated.

sparkWriteAudio( )

```
int sparkWriteAudio( int TrackNo, int StartIndex, int
LengthSamples, void * Buffer )
```

sparkWriteAudio( ) writes samples into an opened audio element. **TrackNo** is the number of the track where the opened audio element to write in can be found. **TrackNo** allowable values range from 1 to **sparkAudioMaxOutputTracks( )** return value.

StartIndex is the index of the first sample to write in the opened audio element. **LengthSamples** is the number of samples to write. If the interval requested falls out of the audio element's limits, out of bound samples are ignored. **Buffer** is a pointer to a memory area at least **LengthSamples*sparkSampleWidth( )** bytes long. It is only possible to write samples in an element once it has been allocated and opened with **sparkNewAudio( )**. Data transfer is optimized when **StartIndex** and **LengthSamples** are multiples of **sparkAudioBlockSize( ) / sparkSampleWidth( )** and **Buffer** refers to memory aligned on a page size boundary. This function is disabled if the Sparks plug-in is called from Batch.

sparkWriteAudio( ) returns the number of samples written.

sparkTruncateAudio( )

```
int sparkTruncateAudio( int TrackNo, int LengthSamples )
```

sparkTruncateAudio( ) changes the number of samples of an opened audio element. **TrackNo** is the number of the track where the opened audio element to truncate can be found. **TrackNo** allowable values range from `1` to **sparkAudioMaxOutputTracks( )** return value. **LengthSamples** is the new number of samples in the opened audio element. It is only possible to downsize an opened audio element's number of samples. Currently, this function only changes the element's out point; no allocated space on the disk is freed. This function is disabled if the Sparks plug-in is called from Batch.

sparkTruncateAudio( ) returns `SPARK_SUCCESS` on success and `SPARK_FAILURE` on failure.

5 Testing Your Spark Using Burn

Summary

Overview

You can test your Spark using Autodesk® Burn®. Burn is a background rendering application that permits asynchronous background processing and rendering. To test a Spark, you should use Burn in stand-alone mode, a mode that allows an IFFFS application to connect directly to a Render Node.

You can use Burn to process a Batch setup, an EditDesk clip, or a timeline clip on a single Render Node. Typically, you should use existing setups or clips for which input clips are already available.

Testing Sparks Using Burn in Stand-alone Mode

You must configure Burn to run in stand-alone mode in order to test your Spark. When Burn is running in stand-alone mode, it uses the rendering network as follows:

- A single Render Node is designated for rendering jobs.
- The Backburner™ Manager for the rendering network is bypassed.
- The Backburner Server on the designated Render Node is disabled.

To test a Spark using Burn:

1. Configure the software configuration file (*init.cfg*) of the Autodesk application to specify which Burn Render Node to use for testing Sparks. See "Configuring an Autodesk Application for Testing Sparks via Burn" on page 56.
2. Disable the Backburner Server on the Render Node designated for testing Sparks. See "Disabling the Backburner Server and Manager" on page 57.
3. Start Burn on the designated Render Node. See"Starting Burn on the Render Node" on page 57.
4. Start your application with a special environment variable used to run Burn in stand-alone mode and then submit Burn jobs to test your Spark. See "Testing Sparks and Previewing Results" on page 59.

Configuring an Autodesk Application for Testing Sparks via Burn

You must set several keywords in the software initialization configuration file (*init.cfg*) to identify the Burn Render Node designated for Sparks testing.

To configure an Autodesk application for testing Sparks via Burn:

1. Log into the account for your Autodesk application.
2. Open the *init.cfg* file for the application. In a terminal, type:

   ```
   dlcfg
   ```

   The *init.cfg* file appears in a text editor.
3. Scroll through the file and locate the following keywords:

   - `BackburnerManagerHostname`
   - `BackburnerManagerPriority`
   - `BackburnerManagerGroup`
4. Modify these keywords to identify the Burn Render Node designated for Sparks testing:

   - Set the `BackburnerManagerHostname` keyword to the name of the Render Node used to test Sparks.
   - Comment out the `BackburnerManagerPriority` and `BackburnerManagerGroup` keywords by entering a # at the beginning of their lines.

   Setting the keywords in this way configures the application to send Burn jobs directly to a Render Node, rather than to the Backburner Manager for the rendering network. For example, if the name of the Burn Render Node used to test Sparks is *RNode1*, the keywords should be:

```
BackburnerManagerHostname RNode1

#BackburnerManagerPriority 50

#BackburnerManagerGroup <groupname>
```

5. Save and close the *init.cfg* file.

   NOTE: You can create a dedicated *init.cfg* file for testing Sparks and start your application with this file. See the *Autodesk Visual Effects and Finishing Configuration File Reference Guide*.

6. Continue with the procedure "Disabling the Backburner Server and Manager" on page 57.

Disabling the Backburner Server and Manager

To run Burn in stand-alone mode, you must disable the Backburner Server and Manager and start Burn manually on the Render Node designated for Sparks testing as explained in the following procedure. This computer must be the same Render Node as specified in the software configuration file (*init.cfg*) of your Autodesk application.

To disable Backburner Server and Manager:

1. Log in as root on the Linux server and open a terminal.
2. Type:

   `/etc/init.d/backburner stop`

   The following messages appear:

   ```
   Stopping Backburner Server ... OK

   Stopping Backburner Applications ... OK
   ```

3. Continue with the procedure "Starting Burn on the Render Node" on page 57.

Starting Burn on the Render Node

You must start Burn manually on the Render Node designated for Sparks testing. This computer must be the same Render Node as specified in the software configuration file (*init.cfg*) of your Autodesk application. When starting Burn manually, you must specify the test project as well. You can also run the debugger simultaneously when rendering the test project.

To start Burn on the Render Node:

1. Log in as root and open a terminal.
2. Type:

   `cd /usr/discreet/``<product_home>`

where <product_home> is the directory where Burn is installed.

3. Type:

   ./bin/start_burn <project_name> <framestore>**/**<host_name>

   where:

   - <project_name> is the name of the project from which you send Burn jobs.
   - <framestore> is *stonefs* or *stonefs1* .
   - <host_name> is the name of the workstation on which you are running the Autodesk application.

4. Press **ENTER**.

   Burn is initialized and start-up messages appear in the terminal.

   When the message "Startup Complete" appears, you are ready to submit jobs to Burn.

5. Submit a Batch setup, library clip, or EditDesk clip to the Render Node to test your Spark. See "Testing Sparks and Previewing Results" on page 59.

To start Burn with the debugger:

1. Log in as root and open a terminal.
2. Type:

   cd /usr/discreet/<product_home>**/bin**

   where <product_home> is the directory where Burn is installed.

3. Set the debug environment variable for Burn by typing one of the following commands.

Shell	Environment Variable Command
bash	**export BURN_DEBUG1**
csh, tcsh	**setenv BURN_DEBUG1**

4. Set the home environment variable for Burn by typing one of the following commands.

Shell	Environment Variable Command
bash	**export BURN_HOME=/usr/discreet/**<product_home>
csh, tcsh	**setenv BURN_HOME /usr/discreet/**<product_home>

   where <product_home> is the directory where your application is installed (such as flame_2010).

5. Type:

   ./start_burn <project_name> <framestore>**/**<host_name>

where:

- `<project_name>` is the name of the project from which you send Burn jobs.
- `<framestore>` is *stonefs* or *stonefs1*.
- `<host_name>` is the name of the workstation on which you are running the Autodesk application.

6. Submit a Batch setup, library clip, or EditDesk clip to the Render Node to test your Spark. See "Testing Sparks and Previewing Results" on page 59.

Testing Sparks and Previewing Results

To test your Spark, you must start your Autodesk application with a special environment variable and then submit the Burn rendering jobs to the designated Render Node. You can preview the rendering results in the *_Burn_* library for Inferno, Flame, Flare, or Flint, on the Smoke EditDesk, or in the clip library of Smoke.

To test a Spark with Burn:

1. Log into the account for your Autodesk application and then open a terminal.
2. Set the environment variable that configures the application to run Burn in stand-alone mode. In the terminal, type:

   `setenv DL_BURN_STANDALONE_MODE 1`

3. With the terminal still open, start the application by typing its name, for example, **`smoke`**. Do not use the application's desktop shortcut, if one exists.

   The application starts and Burn is run in stand-alone mode. You are ready to create and send Burn jobs from your application to test your Spark.

   NOTE: If you are running the Autodesk application on the O2 platform, start the application using the **`-Z`** option to disable volume integrity check.

4. Load a Batch setup, an EditDesk clip, or a timeline clip in a clip library and click Burn.
5. When prompted, click Confirm to render the setup or clip remotely on the Render Node.

   The job is submitted to the Burn rendering engine on the designated Render Node.

6. Preview the result in the *_Burn_* library for Inferno, Flame, Flare, or Flint, the Smoke EditDesk, or in the clip library of Smoke.
7. When you have finished testing your Spark, type the following command in the terminal to shut down Burn on the designated Render Node:

   `./burnShutDown` `<-dM>`

where:

`-d` is the name of the Render Node; if this variable is not specified, localhost is used by default.

`-M` is the port number used by the Burn rendering engine; if this variable is not specified, 9000 is used by default.

Using a Script for Your Test Setup

If you want to run the same Burn job for several iterations of testing your Spark code, you can use the script *submit_burn*. Use this script exclusively for testing changes during development. You can only use this script after first submitting a Burn job through Batch. This script cannot be used with Autodesk Finishing applications to test a Spark on Burn.

To use the *submit_burn* script:

1. Use Burn to process a Batch setup from Inferno, Flame, or Flint.

   You must first submit a Batch setup to be processed by Burn before you can use the *submit_burn* script.

2. Log in as root on the Linux workstation.

3. Type:

   `cd /usr/discreet/``<product_home>`**`/bin`**

   where `<product_home>` is the directory where Inferno, Flame, or Flint is installed.

4. Type:

   `./submit_burn` `<render_node_name> <job_name> <first_frame> <last_frame>`

   where:

   - `<render_node_name>` is the name of the Render Node designated for Sparks testing.
   - `<job_name>` is the full name of the Burn job, for example, *Burn_clienthost_060620_15.45.33*. This name will appear in the *_Burn_* library when you submit the first rendition of the Burn job. From then on, use the same name to run the script.
   - `<first_frame>` and `<last_frame>` specify the first and last frames to be rendered.

   NOTE: Only use this script when you are working in Burn stand-alone mode.

Building Spark DSO Libraries

When you compile a Sparks DSO library, it is important to maintain Sparks integration flexibility with Autodesk applications. Avoid using explicit dependencies to GL and GLU in the link command during the compilation of the Sparks DSO. For example, do not use the **`ld`** command with the **`-lGL`** or **`-lGLU`** options.

Using Sparks and Reactivating the Distributed Queueing System

When you have fully tested your Spark code, do the following to use your Spark on the Distributed Queueing System:

- Stop the Autodesk application from running in stand-alone mode by typing:

  `unsetenv DL_BURN_STANDALONE_MODE`

- Install the Spark on each Render Node.
- Reset the `BackburnerManagerHostname`, `BackburnerManagerPriority`, and `BackburnerManagerGroup` keywords in the software initialization configuration file (*init.cfg*) for the Autodesk application. See "Configuring an Autodesk Application for Testing Sparks via Burn" on page 56.
- Restart Backburner Manager, if necessary.
- Start Backburner Server on each Render Node on the Distributed Queueing System. See "Starting Burn on the Render Node" on page 57.

Index

U

V