OpenWF Composition Specification
Version 1.0
5 November 2009

Editor: Robert Palmer

Contents

1 Overview

This document details the programming interface to OpenWF Composition. Developed as an open standard by The Khronos Group, OpenWF Composition serves as a low-level interface for two-dimensional composition used in embedded and/or mobile devices.

The main goal of OpenWF Composition is to provide a device-independent and vendor-neutral interface to allow compositing window systems to take advantage of hardware acceleration.

This document defines the C language binding to OpenWF Composition. Other language bindings may be defined by Khronos in the future. We use the term "implementation" to refer to the software and/or hardware that implements OpenWF Composition functionality, and the term "user" to refer to any software that makes use of OpenWF Composition.

1.1 What is Composition?

Composition is the process of combining multiple content sources together into a single image. The process may involve some transformation of the source such as scaling and rotation. Transparency information can be used to shape content and to allow it to appear blended over other content.

It is common for composition to be the final stage of the graphics pipeline before physical display. Some silicon vendors include composition capabilities in their display controller hardware.

1.2 What is a Compositing Window System?

Conventional window systems involve some form of arbitration of access to a single buffer associated with a display. If an application's window is partially obscured by another window, the window system provides the means to ensure the obscured area is not rendered.

In a compositing window system, window rendering can be directed to individual off-screen buffers that can be rendered without regard to the visibility of the associated window. The window system, or some privileged delegate, is then responsible for compositing the window's off-screen buffer together with the contents of other windows so that it can be displayed.

The benefit of redirecting window rendering to off-screen buffers is that it permits flexible post-processing of window contents. Arbitrary windows can have visual effects, such as transparency, applied to them without involving the applications that are rendering them.

1.3 Target Users

Two classes of users were considered when defining the requirements for the design of the OpenWF Composition API. Conventional applications are not intended to be direct users.

Window Systems
OpenWF Composition must provide services to enable Window Systems to generically and efficiently achieve artifact-free composition of multiple graphical sources including video, 3D and 2D content types. It must be suitable for displaying the composition results on-screen as well as using the results for non-displayable use-cases such as providing input into a video stream encoder or saving a screen-shot to disk.

System Integrators
OpenWF Composition may be used by OEMs to assist the integration of pure composition hardware into proprietary Windowing System driver implementations. It must be possible to use OpenWF Composition to abstract a single composition device that is decoupled from the display controller. The API should minimize the effort required to integrate the composition driver with the display controller driver whilst still enabling optimal data-paths.

Additionally, platform vendors may make OpenWF Composition available to privileged clients of the Window System such as client-side Window Managers.

1.4 Target Hardware

OpenWF Composition is designed to be implementable on a wide range of graphics hardware, but is mainly targeted at low-power 2D composition hardware on systems where memory bandwidth may be a limiting factor. It should be possible to implement OpenWF Composition on any device that is capable enough to support OpenGL ES 1.1.

OpenWF Composition is designed to abstract implementations that make use of multiple hardware devices to achieve composition. On hardware platforms where multiple devices are provided by one vendor, the vendor may provide a single implementation that transparently achieves optimal division of workload between the available devices.

On hardware platforms where an OEM integrates multiple devices from independent vendors, OpenWF Composition implementations can exist that drive a single device. In this case the user, which may be some OEM software, is responsible for the delegation of composition operations to the available devices.

Devices can be chained together such that the output of one device becomes the input of another device. For example the output of an OpenWF Composition device may be used as input to an OpenWF Display device. The interfaces allow zero-copy transfers using optimal synchronization.

1.5 Design Philosophy

OpenWF Composition is intended to provide a hardware abstraction layer that will allow accelerated performance on a variety of hardware platforms. Functions that are not expected to be amenable to hardware acceleration in the near future were either not included or defined to be optional.

All graphical and multimedia content on a system must be able to be composited and displayed using OpenWF Composition, including critical graphics that must not fail[1]. All rendering operations make use of implementation-owned objects referenced using opaque handles. This allows the implementation to fully allocate resources at object creation time so that rendering will not fail due to insufficient memory.

OpenWF Composition depends on EGL for synchronization primitives and optionally for content holders. This allows reuse of existing Khronos infrastructure and ensures that all Khronos media APIs benefit from the continued improvement of this central functionality.

Complex composition operations can be achieved using the image input mechanisms of existing rendering APIs such as OpenGL ES and OpenVG. OpenWF Composition does not attempt to match this level of functionality and instead abstracts a simpler form of composition that can be implemented using less silicon, power and memory bandwidth than the existing APIs.

[1] An example usage of critical graphics would be displaying the digits entered by a user making an emergency telephone call on a mobile device.

2 Definitions, Data Types, Etc.

OpenWF Composition type definitions and function prototypes are found in a `wfc.h` header file, located in a WF subdirectory of a platform-specific header file location.

2.1 Versioning

The `<WF/wfc.h>` header file defines constants indicating the version of the specification. Future versions will continue to define the constants for all previous versions with which they are backward compatible.

For the current specification, the constant `OPENWFC_VERSION_1_0` is defined. The version may be queried at runtime using the **wfcGetStrings** function (see Section 10.3.2).

```
#define OPENWFC_VERSION_1_0 (1)
```

2.2 Primitive Data Types

Composition defines a number of primitive data types by means of C `typedef`s. The actual data types used are platform-specific.

WFCboolean

`WFCboolean` is an enumeration that only takes on the values of `WFC_FALSE` (0) or `WFC_TRUE` (1). Any non-zero value used as a `WFCboolean` will be interpreted as `WFC_TRUE`. If `<KHR/khrplatform.h>` is available [KHR09], the boolean enumerations will be equated to the corresponding values of the `khronos_boolean_enum_t`.

```
typedef enum {
    WFC_FALSE = KHRONOS_FALSE,
    WFC_TRUE  = KHRONOS_TRUE
} WFCboolean;
```

WFCint

`WFCint` defines a 32-bit two's complement signed integer. If `<KHR/khrplatform.h>` is available, `WFCint` will be defined as `khronos_int32_t`.

WFCfloat

`WFCfloat` defines a 32-bit IEEE 754 floating-point value. If `<KHR/khrplatform.h>` is available, `WFCfloat` will be defined as `khronos_float_t`.

WFCbitfield
`WFCbitfield` defines a 32-bit unsigned integer value, used for parameters that may combine a number of independent single-bit values. A `WFCbitfield` must be able to hold at least 32 bits. If `<KHR/khrplatform.h>` is available, `WFCbitfield` will be defined as `khronos_uint32_t`.

In order to ensure that applications continue to run correctly on Composition implementations that contain extensions or that implement future versions of this specification, `WFCbitfield` arguments should have their unused bits set to zero.

All pointer arguments must be aligned according to their datatype, e.g., a WFCfloat * argument must be a multiple of 4 bytes.

2.3 Floating-Point and Integer Representations

All floating-point values are specified in standard IEEE 754 format. However, implementations may clamp extremely large or small values to a restricted range and internal processing may be performed with lesser precision.

Handling of special values is as follows. Positive and negative 0 values must be treated identically. Values of +Infinity, -Infinity, or NaN (not a number) yield unspecified results. Denormalized numbers may be truncated to 0. Passing any arbitrary value as input to any floating-point argument must not lead to process termination.

Also, the following definitions denote the standard range of integer and floating-point values:

WFC_MAX_INT
The macro `WFC_MAX_INT` defines the largest positive integer value that will be accepted by an implementation. `WFC_MAX_INT` is defined to be 2^{24}, or 16,777,216. The smallest negative integer value that will be accepted by an implementation is given by (−`WFC_MAX_INT`), or -16,777,216.

WF_MAX_FLOAT
The macro `WFC_MAX_FLOAT` defines the largest positive floating-point value that will be accepted by an implementation. `WFC_MAX_FLOAT` is defined to be 2^{24}, or 16,777,216. The smallest negative floating-point value that will be accepted by an implementation is given by (−`WFC_MAX_FLOAT`), or -16,777,216.

2.4 Colors

Colors in OpenWF Composition other than those stored in image pixels (e.g., colors for clearing) are represented as non-premultiplied (see Section 2.4.1) color

values. An implementation may support source and target images defined in a number of color spaces, including sRGB, linear RGB, linear grayscale (or luminance) and non-linearly coded, perceptually-uniform grayscale, in premultiplied or non-premultiplied form.

Color and alpha values lie in the range [0,1] unless otherwise noted. This applies to intermediate values in the pixel pipeline as well as to application-specified values. If an alpha channel is present but has a bit depth of zero, the alpha value of each pixel is taken to be 1.

2.4.1 Premultiplied Alpha

In premultiplied alpha (or simply premultiplied) formats, a pixel (R, G, B, α) is represented as (α*R, α*G, α*B, α). Alpha is always coded linearly, regardless of the color space. The terms associated alpha and premultiplied alpha are synonymous.

If a premultiplied image contains pixels whose color values exceed their associated alpha value, those pixels are considered to have undefined values. This condition must not lead to process termination.

2.4.2 Color Format Conversion

Color values are converted between different formats and bit depths as follows. First, premultiplied color values with non-zero alpha values are divided out to obtain a non-premultiplied representation for the color. It is recommended that premultiplied color values are clamped to their corresponding alpha value prior to the division.

If the source and destination formats differ only in the number of bits per color channel, each channel value is multiplied by the quotient $(2^d - 1)/(2^s - 1)$ (where d is the number of bits in the destination channel and s is the number of bits in the source channel) and rounded to the nearest integer.

Approximations may be used in place of exact multiplication. Approximations must lie strictly within $1/(2^d - 1)$ of the value that would be produced by exact multiplication. Converting from a lesser to a greater number of bits and back again must result in an unchanged value.

If the destination format has stored alpha, the previously saved alpha value is stored into the destination. If the destination format has premultiplied alpha, each color channel value is multiplied by the corresponding alpha value. It is recommended that the premultiplied color values are clamped between 0 and their corresponding alpha value.

2.5 Enumerated Data Types

A number of data types are defined using the C `enum` keyword. In all cases, this specification assigns each enumerated constant a particular integer value. Extensions to the specification wishing to add new enumerated values must register with the Khronos Group to receive a unique value (see Section 10.2).

Applications making use of extensions should cast the extension-defined integer value to the proper enumerated type.

The enumerated types (apart from `WFCboolean`) defined by Composition are:

- `WFCContextAttrib`
- `WFCContextType`
- `WFCDeviceAttrib`
- `WFCDeviceClass`
- `WFCErrorCode`
- `WFCFiltering`
- `WFCRotation`
- `WFCStringID`
- `WFCTransparencyType`

2.6 Handle-based Data Types

Objects are accessed using opaque handles. The use of handles allows these potentially large and complex objects to be stored under API control. For example, they may be stored in special memory and/or formatted in a way that is suitable for use by a hardware implementation.

Handles make use of the `WFCHandle` data type. For reasons of binary compatibility between different Composition implementations on a given platform, a `WFCHandle` is defined as a 32-bit unsigned integer value.

Handles to distinct objects of each type must compare as unequal using the C == operator.

The `WFCHandle` subtypes defined in the API are:

- `WFCDevice` – a reference to a device (see Section 3)
- `WFCContext` – a reference to a destination-specific context (see Section 5)
- `WFCSource` – a reference to a source image provider (see Section 6.1)
- `WFCMask` – a reference to a mask image provider (see Section 6.2)
- `WFCElement` – a reference to a scene specification element (see Section 7)

The symbol `WFC_INVALID_HANDLE` represents an invalid `WFCHandle` that can be passed to certain functions and is used as an error return value from functions that return a `WFCHandle` subtype.

```
#define WFC_INVALID_HANDLE ((WFCHandle)0)
```

2.7 EGL Handles

Handles to EGL resources must be cast into the corresponding OpenWF Composition type before using them with this API. Such types are not subtypes of `WFCHandle`.

A `WFCEGLDisplay` is an opaque handle to an `EGLDisplay` created using the EGL.

A `WFCEGLSync` is an opaque handle to an `EGLSyncKHR` created using the EGL. Refer to [EGL08] for details on an EGLSync.

```
typedef void* WFCEGLSync;
```

2.8 Streams

OpenWF Composition uses *streams* for both input and output of pixel data. A stream is a container for multiple image buffers that share the same properties. Streams are used to allow renderers to pass image data to consumers of such data. Transferring image data from an OpenGLES renderer into OpenWF Composition is one example of using streams.

Streams encapsulate the queuing state of the internal buffers. The user is responsible for connecting producers and consumers to the stream. Once connected, *autonomous* renderers, like OpenWF Composition, can send and receive frames without user interaction.

Support for particular stream formats can vary between OpenWF Composition devices. Streams may be simultaneously connected as inputs to multiple Contexts. Individual streams may have limits on the number of Contexts that may be simultaneously connected as consumers.

Creation of streams is outside of the scope of OpenWF Composition. Functions that make use of streams use the platform-specific type `WFCNativeStreamType`, which is the type of a handle to the platform's representation of a stream.

```
typedef <platform-specific> WFCNativeStreamType;
```

2.8.1 Functional Requirements

All stream implementations must support the following functional requirements to be usable with OpenWF Composition.

Creation
It must be possible to create single- and multi-buffered streams that can be bound as:

- WFCSource inputs
  - With at least 8-bits per-channel RGBA data per-pixel, where channel order is not significant[2]
- WFCMask inputs
  - With at least 8-bits of alpha data per-pixel
- WFCContext destinations
  - With at least 8-bits per-channel RGBA data per-pixel, where channel order is not significant

It must be possible to create such streams with dimensions of 128 by 128 pixels[3].

Content Provision
When a stream is connected as an input to a Context, the stream user must be able to temporarily gain exclusive access to a 2D image "backbuffer" into which color data can be written. It must be possible to submit this backbuffer into the stream so that the Context observes the contents of the backbuffer when it next renders frames in which the stream is visible. Context rendering must not be affected by the stream user gaining and maintaining exclusive access to the backbuffer. Submission of the backbuffer into the stream must not affect the rendering of any frame that began before the submission.

Provocation of Event-Driven Autonomous Composition
If a Context is performing autonomous rendering (see Section 8) and an input stream is connected such that it affects the rendering results of the Context, the submission of a backbuffer into that stream must cause the Context to render a new frame that includes the backbuffer's contents in finite time[4]. The Context may choose to skip processing of the backbuffer if another backbuffer is submitted into the stream prior to the Context starting to render.

Content Notification
When a stream is connected as a Context's destination, the stream user must be able to register for notification of an update to the stream's contents caused by the Context rendering a new frame.

---

[2] Note that this does not preclude the stream implementation and OpenWF Composition also supporting streams with fewer bits per-channel, e.g. RGB565.
[3] It is expected that most implementations will support streams with a range of sizes. However the Conformance Tests only require streams to be 128x128.
[4] This implies that if there are multiple Contexts repeatedly reading from the same Stream, the Contexts must be able to read from different stream buffers. Otherwise continual overlapping reads from the Contexts could prevent a submitted backbuffer being processed in finite time.

Content Extraction
When a multi-buffered stream is connected as a Context's destination, the stream user must be able to temporarily gain exclusive access to a 2D image "frontbuffer" containing the color data of the most recently rendered frame. Rendering operations in progress must not delay the stream user's access to this frontbuffer.

Synchronization
The stream interface must allow the stream user to display images at specific times, subject to a degree of tolerance. This is necessary for achieving audio-visual synchronization. The tolerances and mechanisms for achieving this synchronization are not specified by OpenWF Composition.

Threading
It must be possible to create and use streams with OpenWF Composition Contexts such that the stream user operates in a separate thread to the user of the OpenWF Composition Context.

Immutability
The following properties of streams used with OpenWF Composition must not change:

- The dimensions of the stream
- The format of the stream

2.9 Attribute Lists

Most object creation functions accept an attribute list parameter to allow the API to be extended without the need for new entry points. These lists share a common format and behavior.

The list is specified by a `const WFCint *attribList` parameter. All attribute names in `attribList` are immediately followed by the corresponding value. The list is terminated with `WFC_NONE`. `attribList` may be `NULL` or empty (first attribute is `WFC_NONE`). Providing a non-`NULL` `attribList` that is not terminated with `WFC_NONE` will result in undefined behavior. Unspecified attributes assume their default values.

The set of valid attributes is specified with the definition of each function. Extensions may define additional attributes and corresponding legal values. If `attribList` contains an invalid attribute, a `WFC_ERROR_BAD_ATTRIBUTE` error will be generated. If `attribList` contains an invalid attribute value for a legal attribute, a `WFC_ERROR_ILLEGAL_ARGUMENT` error will be generated.

```
#define WFC_NONE (0)
```

2.10 Coordinate Systems

Destination rectangles are specified in the *context coordinate system*. This is a pixel-based coordinate scheme that, for on-screen output, is equivalent to the screen coordinates subjected to a user-specified Context rotation. For off-screen output, a user-specified Context rotation maps between the *context coordinate system* and the destination *image coordinate system*.

The coordinate systems are oriented such that values along the X axis increase from left to right and values along the Y axis increase from top to bottom[5]. A change of 1 unit along an axis corresponds to moving by one pixel.

Source rectangles are specified in a pixel-based *image coordinate system* relative to top left of the image they are associated with. Sub-pixel coordinate values are supported for source rectangles, as described in Section 7.1.3.

2.11 Errors

Where possible, when an API function fails it has no effect. An error condition within an API function must never result in process termination, with the exception of illegal memory accesses caused by the user providing invalid pointers.

Unless otherwise specified, any value returned from a function following an error is undefined. Functions that return handles are one such exception and signal errors by returning `WFC_INVALID_HANDLE` instead of the requested handle. Most API functions do not return an indicator of success or failure; errors are stored in the Device associated with the function call and may be retrieved by calling the **wfcGetError** function described below. If an invalid Device handle is passed to any function, the function has no effect and no error is stored.[6]

The error codes that can be generated are listed in the `WFCErrorCode` enumeration.

[5] Note that this Y axis convention is the opposite of that used by OpenGL and OpenVG.

[6] Despite no error being stored, the user is still able to discover that the device handle is invalid when they pass the device handle to wfcGetError.

```
typedef enum {
    WFC_ERROR_NONE              = 0,
    WFC_ERROR_OUT_OF_MEMORY     = 0x7001,
    WFC_ERROR_ILLEGAL_ARGUMENT = 0x7002,
    WFC_ERROR_UNSUPPORTED       = 0x7003,
    WFC_ERROR_BAD_ATTRIBUTE     = 0x7004,
    WFC_ERROR_IN_USE            = 0x7005,
    WFC_ERROR_BUSY              = 0x7006,
    WFC_ERROR_BAD_DEVICE        = 0x7007,
    WFC_ERROR_BAD_HANDLE        = 0x7008,
    WFC_ERROR_INCONSISTENCY     = 0x7009
} WFCErrorCode;
```

Error codes are retrieved by calling **wfcGetError**.

```
WFCErrorCode wfcGetError(WFCDevice dev);
```

wfcGetError returns the oldest error code provided by an API call on *dev* since the previous call to **wfcGetError** on that device (or since the creation of the device). No error is indicated by a return value of 0 (`WFC_ERROR_NONE`). After the call, the error code is cleared to 0. If *dev* is not a valid Device, `WFC_ERROR_BAD_DEVICE` is returned. The possible errors that may be generated by each API function are shown below the definition of the function.

2.12 Setting and Querying Attributes

Many areas of this specification require the setting and/or querying of values from a list of attributes. There are four variants of these attribute get and set functions. The variants are differentiated by a suffix: **i** for integral values, **f** for floating-point values, and **iv** and **fv** for vectors of integers and floating-point values, respectively.

For example, to query a `WFCint`, `WFCbitfield`, `WFCboolean`, enumeration, or other integral type attribute from an Element, **wfcGetElementAttribi** would be used. In order to query a `WFCfloat` type attribute from an Element, **wfcGetElementAttribf** would be used.

The accessors that may be used with each attribute are stated alongside the definition of the attribute. Attempting to read or write an attribute with an accessor that is not permitted by the attribute will result in a `WFC_ERROR_BAD_ATTRIBUTE` error.

When setting an integral type value using the floating-point function (ending with **f** or **fv**), or retrieving a floating-point value using an integer function (ending with **i** or **iv**), the value is converted to an integer using a mathematical

floor operation. If the resulting value is outside the range of integer values, the closest valid integer value is submitted. Similarly, when setting a floating-point value using the integer function or retrieving an integer value using a floating-point function, if there is any loss of precision, the closest floating-point value is submitted. Attributes that do not obey the default integer-float conversion may state their own conversion behavior.

All array variants (ending with **iv** or **fv**) are fixed length, either by attribute definition or via a corresponding length attribute. The `count` parameter used by the array variant functions must match the length of array being accessed.

Certain attributes are read-only. Attempting to set a read-only attribute has no effect other than generating a `WFC_ERROR_BAD_ATTRIBUTE` error.

If the variant of the query function matches the variant of the set function previously used on a handle, the original value (except as specifically noted) is returned by the query function, even if the implementation makes use of a truncated or quantized value internally. This rule ensures that state may be saved and restored without degradation.

2.13 Contexts and Threading

OpenWF Composition has been designed such that all API resources are identified by thread-independent handles. There is no thread-specific state. API handles can be used safely across multiple threads. All of the API functions are thread safe.

3 Composition Pipeline

This section defines the OpenWF Composition pipeline mechanism by which scene Elements are rendered. In the course of producing a frame, implementations are not required to match the stated pipeline stage-for-stage; they may take any approach to rendering so long as the final results match the results of the stated pipeline within the tolerances defined by the conformance testing process.

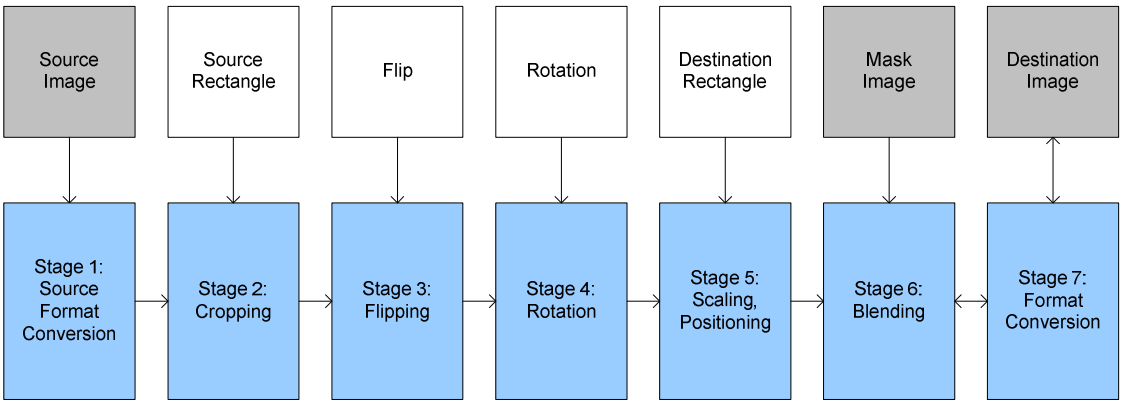


Figure 1 - The Composition Pipeline

When rendering a frame, the destination is first cleared to a user specified background color before any scene Elements are rendered using the Composition pipeline.

Stage 1: Source Format Conversion
The color data from the source image is converted into RGBA form. The alpha format of the source is not modified.

Stage 2: Cropping
Cropping extracts a sub-rectangle of the source image. The user specifies this rectangle via an X-offset, Y-offset, width and height.

Stage 3: Flipping
If the user enables flipping, the output of the cropping stage is flipped vertically, inverting the image top-to-bottom.

Stage 4: Rotation
The output of the flipping stage is rotated clockwise by a user-specified 90-degree increment.

Stage 5: Scaling, Positioning
The output of the rotation stage is scaled and offset according to a user-specified destination rectangle. Color and alpha values are computed for destination pixels by filtering image values from the previous stage. The user has some control over the class of filtering that is used. Filtering is performed in the alpha format of the source.

Stage 6: Blending
The output from the previous stage is blended with the destination according to the user-specified blending rules. The blending can involve per-pixel opacity defined in the source image or separately supplied mask image in addition to an opacity value applied to the entire Element. Blending produces a premultiplied result.

Stage 7: Format conversion
The output of blending is converted into the destination format prior to storage. This stage also provides destination pixel values back to the previous stage converted into the RGBA form for use in blending.

4 Devices

All composition operations and resources are ultimately associated with a `WFCDevice`. A `WFCDevice` is an abstract device that is capable of performing composition operations. Commonly a `WFCDevice` will correspond to a unit of graphics hardware. Devices can vary in their support for specific input and output formats.

Each device maintains an error state, as described in Section 2.11.

4.1 Device Attributes

A Device consists of attributes enumerated by `WFCDeviceAttrib`.

```
typedef enum{
    WFC_DEVICE_CLASS = 0x7030,
    WFC_DEVICE_ID    = 0x7031
} WFCDeviceAttrib;
```

Default device attribute values are shown in ***Table 1***.

Attribute	Type	R/W	Default
WFC_DEVICE_CLASS	WFCDeviceClass	R	N/A
WFC_DEVICE_ID	WFCint	R	N/A

Table 1 – Default device attributes

4.1.1 Device Class

Composition devices fall into two classes; Fully-Capable and Off-Screen Only[7]. They are differentiated by their ability to render to on-screen destinations.

The `WFCDeviceClass` enumeration defines values for each device class.

```
typedef enum {
    WFC_DEVICE_CLASS_FULLY_CAPABLE     = 0x7040,
    WFC_DEVICE_CLASS_OFF_SCREEN_ONLY   = 0x7041
} WFCDeviceClass;
```

Fully-Capable
A fully-capable device has the ability to output to at least one screen as well as to off-screen targets represented by `WFCNativeStreamType`s.

[7] There is no "On-Screen Only" device class. Hardware that can only render to on-screen destinations is catered for by the OpenWF Display specification.

Off-Screen Only
An "off-screen only" device has the ability to compose to off-screen targets represented by `WFCNativeStreamType`s.

Accessors: **wfcGetDeviceAttribi**

4.1.2 Device ID

The `WFC_DEVICE_ID` attribute denotes the ID of the device. This value will not be `WFC_DEFAULT_DEVICE_ID`.

Accessors: **wfcGetDeviceAttribi**

4.2 Enumerating Devices

Devices are identified using integer IDs assigned by the implementation[8]. The number and IDs of the available devices on the system can be retrieved by calling **wfcEnumerateDevices**.

```
WFCint wfcEnumerateDevices(WFCint       *deviceIds,
                           WFCint        deviceIdsCount
                           const WFCint *filterList);
```

The user provides a buffer to receive the list of available device IDs. *`deviceIds`* is the address of the buffer. *`deviceIdsCount`* is the number of device IDs that can fit into the buffer.

The *`filterList`* parameter contains a list of filtering attributes which are used to control the device IDs returned by **wfcEnumerateDevices**. All attributes in *`filterList`* are immediately followed by the corresponding value. The list is terminated with `WFC_NONE`. *`filterList`* may be `NULL` or empty (first attribute is `WFC_NONE`), in which case all valid device IDs for the platform are returned. Providing a non-`NULL` *`filterList`* that is not terminated with `WFC_NONE` will result in undefined behavior. Providing an invalid filter attribute or filter attribute value will result in an empty device ID list being returned. See Section 4.2.1 for the list of available device filtering attributes.

If *`deviceIds`* is `NULL`, the total number of available devices on the system is returned and the buffer is not modified.

If *`deviceIds`* is not `NULL`, the buffer is populated with a list of available device IDs. No more than *`deviceIdsCount`* will be written even if more are available.

[8] IDs do not need to be hard-coded in the implementation and may be generated upon first use if the system supports discovery of devices.

The number of device IDs written into `deviceIds` is returned. If *`deviceIdsCount`* is negative, no device IDs will be written.

Device IDs should not be expected to be contiguous. The list of device IDs will not include a device ID equal to `WFC_DEFAULT_DEVICE_ID`. The list of available devices is not expected to change.

4.2.1 Enumeration Filtering Attributes

The **WFCDeviceFilter** enumeration contains the valid filtering attributes. These attributes are described in sub-sections below.

```
typedef enum {
    WFC_DEVICE_FILTER_SCREEN_NUMBER = 0x7020
} WFCDeviceFilter;
```

The data types and default values for all device filter attributes are listed in the table below.

Attribute	Type	Default
`WFC_DEVICE_FILTER_SCREEN_NUMBER`	`WFCint`	`N/A`

Table 2 – Default device filter attributes

4.2.1.1 Screen Number Filter

The `WFC_DEVICE_FILTER_SCREEN_NUMBER` filter attribute is used to limit the returned device ID list to only include devices that can create on-screen Contexts using the given screen number. If this attribute appears more than once in the filter list, no device IDs will be returned.

4.3 Creating Devices

A device can be created and a handle to it returned by calling **wfcCreateDevice**.

```
WFCDevice wfcCreateDevice(WFCint        deviceId,
                          const WFCint *attribList);
```

The value of *`deviceId`* should be either a device ID retrieved using **wfcEnumerateDevices** or `WFC_DEFAULT_DEVICE_ID`. If *`deviceId`* is `WFC_DEFAULT_DEVICE_ID`, a default device is returned. The system integrator will determine the default device.

```
#define WFC_DEFAULT_DEVICE_ID (0)
```

The *`attribList`* parameter is defined in section 2.9. No valid attributes are defined for *`attribList`* in this specification.

If no device matching *deviceId* exists or if there not enough memory to create the device, WFC_INVALID_HANDLE is returned.

Creation of multiple devices using the same device ID is supported. Each of these devices will have a unique handle. State is not shared between these devices.

4.4 Querying Device Attributes

To query an attribute associated with a device call **wfcGetDeviceAttribi**.

```
WFCint wfcGetDeviceAttribi(WFCDevice       dev,
                           WFCDeviceAttrib attrib);
```

On success, the value of *attrib* for *dev* is returned. Refer to Section 4.1 for a list of valid attributes.

ERRORS

WFC_ERROR_BAD_ATTRIBUTE
- if *attrib* is not a valid device attribute

4.5 Destroying Devices

A device is destroyed by calling **wfcDestroyDevice**.

```
WFCErrorCode wfcDestroyDevice(WFCDevice dev);
```

All resources owned by *dev* are deleted before this call completes. *dev* is no longer a valid device handle. All resource handles associated with *dev* become invalid. Any references held by *dev* on external resources are removed.

All pending composition operations associated with *dev* are completed before this call completes.

If *dev* is not a valid Device, WFC_ERROR_BAD_DEVICE is returned. Otherwise, WFC_ERROR_NONE is returned.

5 Contexts

All composition operations make use of a `WFCContext`. A `WFCContext` stands for a visual scene description applied to either an on-screen or off-screen target. In this scope it represents the state required for a device to be used for composition of a scene.

Composition operations are not serialized across Contexts. This allows independent workloads to be processed in parallel if the hardware platform supports it. If the user requires a particular ordering of operations that are executed on different Contexts, synchronization functions must be used (see Section 9).

A Context is permanently bound to a particular target. All devices must support the creation of multiple off-screen Contexts. Multiple Contexts may share the same target but only one Context may render to the target at a time. There is no concept of a Context being "current" to a thread. Multiple Contexts can be used by a single thread.

A Context's target may be multi-buffered though this is abstracted from the user. Buffer swapping activities such as render target selection and frame completion signaling are performed automatically.

A Context's *scene* consists of the Elements that have been inserted into the scene as well as the Context's attributes. Changes to the scene made by the user do not take effect until the user calls **wfcCommit**.

Upon creation, a Context is considered *inactive*. In this state, composition is user-driven, meaning that rendering will only occur in response to a call to **wfcCompose**. If a Context is made *active* using **wfcActivate**, rendering will occur automatically using the Context's most recently committed scene.

5.1 Context Attributes

A Context consists of attributes enumerated by `WFCContextAttrib`.

```
typedef enum {
    /* Read-only */
    WFC_CONTEXT_TYPE              = 0x7051,
    WFC_CONTEXT_TARGET_HEIGHT     = 0x7052,
    WFC_CONTEXT_TARGET_WIDTH      = 0x7053,
    WFC_CONTEXT_LOWEST_ELEMENT    = 0x7054,

    /* Read-write */
    WFC_CONTEXT_ROTATION          = 0x7061,
    WFC_CONTEXT_BG_COLOR          = 0x7062,
} WFCContextAttrib;
```

Default Context attribute values are shown in ***Table 3***.

Attribute	Type	R/W	Default
WFC_CONTEXT_TYPE	WFCContextType	R	N/A
WFC_CONTEXT_TARGET_HEIGHT	WFCint	R	N/A
WFC_CONTEXT_TARGET_WIDTH	WFCint	R	N/A
WFC_CONTEXT_LOWEST_ELEMENT	WFCElement	R	N/A
WFC_CONTEXT_ROTATION	WFCRotation	R/W	WFC_ROTATION_0
WFC_CONTEXT_BG_COLOR	WFCfloat	R/W	(0, 0, 0, 1)

Table 3 – Default context attributes

5.1.1 Context Type

The `WFCContextType` enumeration defines values for the types of Contexts.

```
typedef enum {
    WFC_CONTEXT_TYPE_ON_SCREEN  = 0x7071,
    WFC_CONTEXT_TYPE_OFF_SCREEN = 0x7072
} WFCContextType;
```

A Context's type is immutable, determined by the function used to create it.

Accessors: **wfcGetContextAttribi**

5.1.2 Target Size

A Context's target width and height is the size in pixels of the destination area that can be rendered. This is the area into which Composition Elements are specified. For on-screen composition, this value will depend upon the current display mode of the associated display. In the case of off-screen composition to a stream, this size will be equal to the dimensions of the stream.

Accessors: **wfcGetContextAttribi**

5.1.3 Lowest Element

The lowest Element (see Section 7) inside the Context's scene can be queried via the `WFC_CONTEXT_LOWEST_ELEMENT` attribute. The value will be `WFC_INVALID_HANDLE` if the Context has no Elements inside its scene.

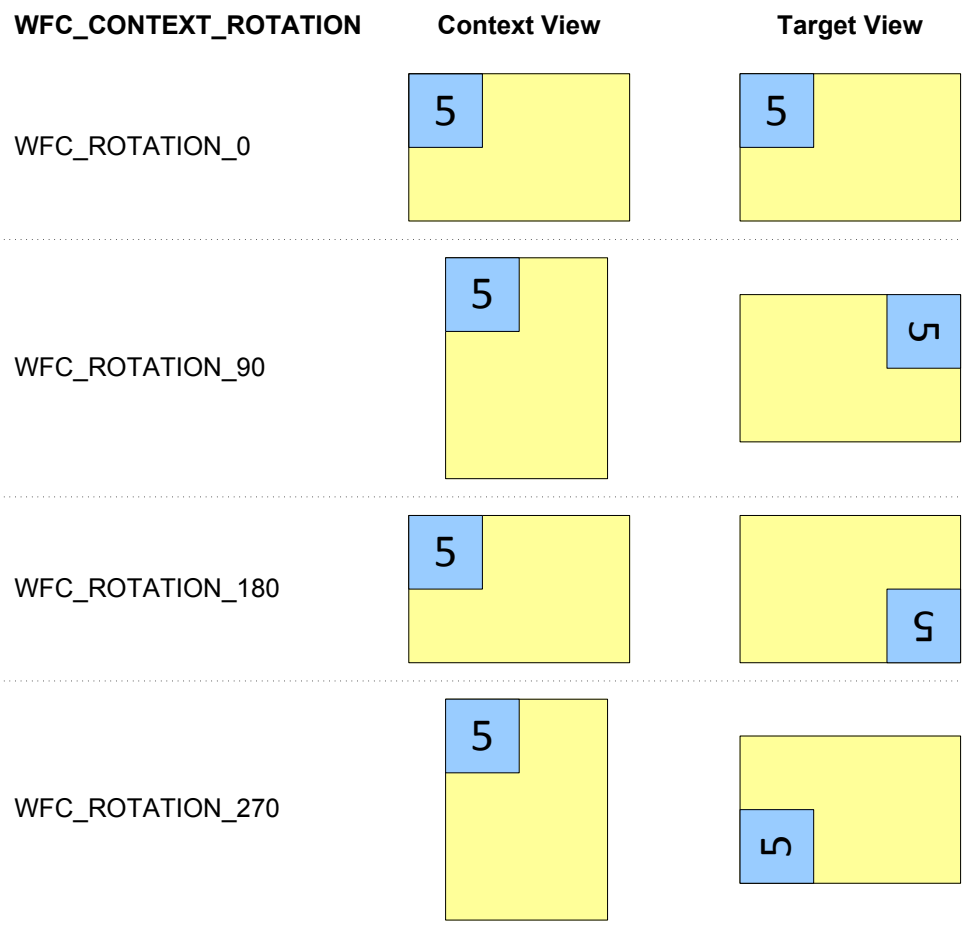
Accessors: **wfcGetContextAttribi**

5.1.4 Rotation

The `WFCRotation` enumeration defines the values for setting rotation. All values represent clockwise rotations.

```
typedef enum {
    /* Clockwise rotation */
    WFC_ROTATION_0   = 0x7081,
    WFC_ROTATION_90  = 0x7082,
    WFC_ROTATION_180 = 0x7083,
    WFC_ROTATION_270 = 0x7084
} WFCRotation;
```

`WFC_CONTEXT_ROTATION` is a mutable attribute that determines the transform between Composition Element destination rectangles and the Context's target.



This value must be a valid `WFCRotation`. Changing the value of this attribute does not affect the Context's reported target size. `WFC_CONTEXT_ROTATION` is independent of any rotation setting made outside this API and is therefore applied in addition to any such rotation[9].

Accessors: **wfcGetContextAttribi, wfcSetContextAttribi**

5.1.5 Background Color

A Context's background color is determined by `WFC_CONTEXT_BG_COLOR`. This is a vector of four components ordered red, green, blue then alpha. Each component ranges between 0 and 1. Attempting to set values outside this range will fail. This color will only contribute to destination areas not covered by an opaque Composition Element. The alpha value must be 1 for on-screen Contexts and off-screen screen Contexts that target images without alpha.

[9] It is expected that a single vendor implementing both a Composition driver and a Display Controller driver could optimise these two logical rotations into a single physical rotation.

If **wfcSetContextAttribi** is used to set this attribute, the value is interpreted as an unsigned 32-bit integer containing 8 bits of red starting at the most significant bit, followed by 8 bits each of green, blue and alpha. Each color and alpha channel value is conceptually divided by 255.0f to obtain a value between 0 and 1.

If **wfcGetContextAttribi** is used to read this attribute, each color channel or alpha value is clamped to the [0, 1] range, multiplied by 255 and rounded to obtain an 8-bit integer; the resulting values are packed into an unsigned 32-bit integer value in the same format used for setting.

Accessors: **wfcGetContextAttribi, wfcSetContextAttribi, wfcGetContextAttribfv, wfcSetContextAttribfv**

5.2 Creating On-screen Contexts

To create a context to control a screen, call **wfcCreateOnScreenContext**.

```
WFCContext wfcCreateOnScreenContext(
                                WFCDevice     dev,
                                WFCint        screenNumber,
                                const WFCint *attribList);
```

If **wfcCreateOnScreenContext** succeeds, a new Context for *screenNumber* is initialized and a handle to it is returned.

screenNumber is an implementation-specific identifier for the particular screen this Context will be used to control. At most one Context per-screen may be created. If *screenNumber* is WFC_DEFAULT_SCREEN_NUMBER, the system's default screen will be selected. The system integrator will determine the meaning of the default screen.

The structure of *attribList* is described in Section 2.9. No valid attributes are defined in this specification.

Creation of an on-screen Context does not cause any change to the state of the physical screen. The first change to the physical screen is caused by the first successful call to **wfcActivate** or **wfcCompose** on the associated Context.

If an error occurs, WFC_INVALID_HANDLE is returned.

ERRORS

`WFC_ERROR_UNSUPPORTED`
- if *`dev`* does not support the creation of an on-screen Context for the specified screen

`WFC_ERROR_OUT_OF_MEMORY`
- if the implementation can not allocate resources for the Context

`WFC_ERROR_IN_USE`
- if a Context has already been created for *`screenNumber`*

`WFC_ERROR_BAD_ATTRIBUTE`
- if *`attribList`* contains an invalid attribute

5.3 Creating Off-screen Contexts

A Device must support rendering to streams with at least 8-bits per-channel RGBA data per-pixel, where channel order is not significant.

To create a Context for compositing into a user-specified image stream, call **wfcCreateOffScreenContext**.

```
WFCContext wfcCreateOffScreenContext(
                            WFCDevice           dev,
                            WFCNativeStreamType stream,
                            const WFCint       *attribList);
```

If **wfcCreateOffScreenContext** succeeds, a new Context is initialized and a handle to it is returned.

stream must be a valid WFCNativeStreamType that identifies the composition target that the returned Context will compose to. The returned Context places a reference on *stream*. *stream* must be compatible as a target with *dev*. Multiple Contexts may be bound to the same stream, but only one Context is allowed to render to a stream at any point in time.

The structure of *attribList* is described in Section 2.9. No valid attributes are defined in this specification.

If an error occurs, WFC_INVALID_HANDLE is returned.

ERRORS

WFC_ERROR_UNSUPPORTED
- if *stream* is not compatible with *dev*

WFC_ERROR_ILLEGAL_ARGUMENT
- if *stream* is not a valid WFCNativeStreamType

WFC_ERROR_OUT_OF_MEMORY
- if the implementation can not allocate resources for the Context

WFC_ERROR_BAD_ATTRIBUTE
- if *attribList* contains an invalid attribute

5.4 Committing Modifications

A Context's scene can only be updated by committing modifications made to its attributes and Elements. Modifications are cached, left uncommitted, until the user explicitly applies them by calling **wfcCommit**. Query functions reflect the cached values rather than the committed values.

```
void wfcCommit(WFCDevice  dev,
               WFCContext ctx,
               WFCboolean wait);
```

If **wfcCommit** succeeds, the current state of *ctx*'s attributes and corresponding Elements will be synchronously captured and then used as the Context's committed scene.

The implementation will test for configuration conflicts before any modifications are committed. If any such conflict exists, the WFC_ERROR_INCONSISTENCY error is generated and the scene is left in the pre-call state.

The previously committed scene and all image providers associated with it may remain in use for a short period of time after **wfcCommit** returns, i.e. until the implementation finishes any pending composition of the previously committed scene.

wfcCommit completes asynchronously. Synchronization functions (see Section 9) must be used if the user needs to know if or when a particular call to **wfcCommit** has completed. Completion means that *ctx* is ready to accept a new call to **wfcCommit**.

If *wait* is WFC_FALSE and ctx is not ready to accept a new call to **wfcCommit**, this call will fail immediately and the committed scene will not be updated. The previous commit will not be affected. If *wait* is WFC_TRUE, **wfcCommit** will not fail due to *ctx* not being ready to accept a new commit[10]. This may entail a delay in the call returning.

ERRORS

WFC_ERROR_INCONSISTENCY
- if an Element associated with *ctx* has a source rectangle that is not fully contained inside the Element's Source

[10] The two types of behaviour aid user flexibility. Users may choose "immediate rejection" if they are controlling multiple contexts from a single thread and do not wish to be stalled for a long period. Users responsible for a single context may benefit from "stall until ready" rather than being forced to poll.

- if an Element associated with *ctx* has a destination rectangle whose size does not match the size of its Mask

WFC_ERROR_BUSY
- if *wait* is WFC_FALSE and ctx is not ready to accept a new call to **wfcCommit**

WFC_ERROR_BAD_HANDLE
- if *ctx* is not a valid Context associated with *dev*

5.5 Querying Context Attributes

Context attributes can be queried by calling:

```
WFCint wfcGetContextAttribi(WFCDevice        dev,
                            WFCContext       ctx,
                            WFCContextAttrib attrib);

void wfcGetContextAttribfv( WFCDevice        dev,
                            WFCContext       context,
                            WFCContextAttrib attrib,
                            WFCint           count,
                            WFCfloat        *values);
```

The *dev* and *ctx* parameters denote the specific Device and Context being queried. The *attrib* parameter denotes the attribute to retrieve and return via the function return value or the *values* parameter. For the vector-based function, *count* denotes the number of values to retrieve and *values* must be an array of *count* elements.

On success, the value(s) of *attrib* for *ctx* is returned via the function return value or the *values* parameter. On failure, *values* in not modified. Refer to Section 5.1 for a list of valid attributes.

Retrieving handle-typed attributes does not affect the lifetime of the handle.

ERRORS

WFC_ERROR_BAD_ATTRIBUTE
- if *attrib* is an invalid attribute
- if *attrib* does not permit the use of this accessor

WFC_ERROR_ILLEGAL_ARGUMENT
- if *count* does not match the size of *attrib*
- if *values* is NULL or not aligned with its datatype

WFC_ERROR_BAD_HANDLE
- if *ctx* is not a valid Context associated with *dev*

5.6 Setting Context Attributes

Context attributes can be set using:

```
void wfcSetContextAttribi( WFCDevice        dev,
                           WFCContext       ctx,
                           WFCContextAttrib attrib,
                           WFCint           value);

void wfcSetContextAttribfv(WFCDevice        dev,
                           WFCContext       ctx,
                           WFCContextAttrib attrib,
                           WFCint           count,
                           const WFCfloat  *values);
```

The *dev* and *ctx* parameters denote the specific Device and Context, respectively, associated with this function call. The *attrib* parameter denotes the attribute being set. For the vector-based function, *count* denotes the number of values provided and *values* must be an array of *count* elements.

On success, the value(s) of *attrib* for *ctx* are set to the specified value(s). On failure, the Context is not modified. Refer to Section 5.1 for a list of valid attributes.

Note that **wfcCommit** must be called on *ctx* before updated values will affect rendering.

ERRORS

WFC_ERROR_BAD_ATTRIBUTE
- if *attrib* is an invalid attribute
- if *attrib* is an immutable attribute
- if *attrib* does not permit the use of this accessor

WFC_ERROR_ILLEGAL_ARGUMENT
- if *count* does not match the size of *attrib*
- if *values* is NULL
- if *values* is non-NULL and not aligned with its datatype
- if the user attempts to set a valid attribute to an invalid value

WFC_ERROR_BAD_HANDLE
- if *ctx* is not a valid Context associated with *dev*

5.7 Destroying Contexts

A Context is destroyed by calling **wfcDestroyContext.**

```
void wfcDestroyContext(WFCDevice  dev,
                       WFCContext ctx);
```

Following this call, all resources owned by `ctx` are marked for deletion as soon as possible. `ctx` is no longer a valid Context handle. All resource handles associated with `ctx` become invalid. Any references held by `ctx` on external resources are removed when deletion occurs.

All pending composition operations associated with `ctx` are completed before this call completes.[11]

ERRORS

`WFC_ERROR_BAD_HANDLE`
- if `ctx` is not a valid Context associated with `dev`

[11] This allows the user to determine when the context has stopped using external resources, such as source and target buffers.

6 Image Providers

Image providers represent content that can be used as input to Composition. Sources and Masks are two types of image providers. Successful creation of an image provider implies the suitability of the stream for the relevant Composition operations, i.e. their usage is guaranteed not to fail due to incompatibilities of memory location or format.

Image providers can be created from streams. The act of deriving an image provider from a stream, as well as the subsequent usage of the image provider within OpenWF Composition, will never modify the pixel data of the stream[12]. Support for discovering stream formats compatible with a Device is outside of the scope of OpenWF Composition.

6.1 Sources

A Source is a supplier of image data that an Element can reference as the primary source of its color data. A Source may contain an alpha channel, the effect of which is determined by the user (see Section 7.1.7).

Devices must support Sources with at least 8-bits per-channel RGBA data per-pixel, where channel order is not significant. It is recommended that both non-premultiplied and premultiplied alpha formats are supported.

6.1.1 Creating Sources

To create a Source, call **wfcCreateSourceFromStream.**

```
WFCSource wfcCreateSourceFromStream(
                            WFCDevice           dev,
                            WFCContext          ctx,
                            WFCNativeStreamType stream,
                            const WFCint       *attribList);
```

If **wfcCreateSourceFromStream** succeeds, a new `WFCSource` is initialized and a handle to it is returned. The returned `WFCSource` places a reference on *`stream`*. The Source is guaranteed to be usable for input to composition with *`ctx`*. If multiple Sources are created from the same stream, each Source will have a unique handle.

[12] This is in contrast to EGLImage extensions such as OES_EGL_image [OES07] in which the act of deriving an API-specific resource causes the pixel data owned by the EGLImage to become undefined. Composition relies on the stream being allocated in a format and location suitable for Composition at creation time.

stream must be a valid stream. *stream* must not be in use as the target of *ctx*. *stream* must be compatible as a Source with *dev*.

The structure of *attribList* is described in Section 2.9. No valid attributes are defined for *attribList* in this specification.

On failure, **wfcCreateSourceFromStream** returns WFC_INVALID_HANDLE.

ERRORS

WFC_ERROR_UNSUPPORTED
- if *stream* is valid but not suitable for composition with *dev*

WFC_ERROR_OUT_OF_MEMORY
- if the implementation fails to allocate resources for the Source

WFC_ERROR_ILLEGAL_ARGUMENT
- if *stream* is not a valid WFCNativeStreamType

WFC_ERROR_BAD_ATTRIBUTE
- if *attribList* contains an invalid attribute

WFC_ERROR_BUSY
- if *stream* can not be used with *ctx* at this time due to other Context's use of *stream*

WFC_ERROR_IN_USE
- if *stream* is in use as the target of *ctx*

WFC_ERROR_BAD_HANDLE
- if *ctx* is not a valid Context associated with *dev*

6.1.2 Destroying Sources

To destroy a composition source, call **wfcDestroySource.**

```
void wfcDestroySource(WFCDevice dev,
                      WFCSource src);
```

Following this call, all resources associated with *src* which were allocated by the implementation are marked for deletion as soon as possible. The reference placed on the Source's stream at creation time is removed when deletion occurs. *src* is no longer a valid Source handle. If *src* is attached to an Element at the time **wfcDestroySource** is called, the Source continues to exist until it is detached or the Element is destroyed.

ERRORS

WFC_ERROR_BAD_HANDLE
- if *src* is not a valid Source associated with *dev*

6.2 Masks

A Mask is a supplier of image data that an Element can reference to mask its output. The Mask is used to determine per-pixel opacity information, the effect of which is determined by the user (see Section 7.1.7).

A Device must support Masks with at least 8-bits of alpha data per-pixel. It is recommended that a 1-bit alpha format is supported. The ideal format uses sequential bytes, 8 pixels per byte, with increasing horizontal positions ordered from LSB to MSB within each byte and each scanline padded to a multiple of 32 bits.

Composition is defined to process Masks in terms of a single-channel. When using a multi-channel format as a Mask, rendering occurs as if the Mask is first converted to single-channel format using the format's conversion rules.

6.2.1 Creating Masks

To create a Mask, call **wfcCreateMaskFromStream.**

```
WFCMask wfcCreateMaskFromStream(
                            WFCDevice           dev,
                            WFCContext          ctx,
                            WFCNativeStreamType stream,
                            const WFCint       *attribList);
```

If **wfcCreateMaskFromStream** succeeds, a new `WFCMask` is initialized and a handle to it is returned. The returned `WFCMask` places a reference on *`stream`*. The source is guaranteed to be usable for input to composition with *`ctx`*. If multiple Masks are created from the same stream, each Mask will have a unique handle.

`stream` must be a valid stream. *`stream`* must not be in use as the target of *`ctx`*. *`stream`* must be compatible as a Mask with *`dev`*.

The structure of *`attribList`* is described in Section 2.9. No valid attributes are defined in this specification.

On failure, **wfcCreateMaskFromStream** returns `WFC_INVALID_HANDLE`.

ERRORS

`WFC_ERROR_UNSUPPORTED`
- if *`stream`* is valid but not suitable for composition with *`ctx's`* Device

`WFC_ERROR_OUT_OF_MEMORY`
- if the implementation fails to allocate resources for the Mask

`WFC_ERROR_ILLEGAL_ARGUMENT`
- if *`stream`* is not a valid `WFCNativeStreamType`

`WFC_ERROR_BAD_ATTRIBUTE`
- if *`attribList`* contains an invalid attribute

`WFC_ERROR_BUSY`
- if *`stream`* can not be used with *`ctx`* at this time due to other Context's use of *`stream`*

`WFC_ERROR_IN_USE`
- if *`stream`* is in use as the target of *`ctx`*

`WFC_ERROR_BAD_HANDLE`
- if *`ctx`* is not a valid Context associated with *`dev`*

6.2.2 Destroying Masks

To destroy a Mask, call **wfcDestroyMask**.

```
void wfcDestroyMask(WFCDevice dev,
                    WFCMask   mask);
```

Following this call, all resources associated with *`mask`* which were allocated by the implementation are marked for deletion as soon as possible. The reference placed on the Mask's stream at creation time is removed when deletion occurs. *`mask`* is no longer a valid Mask handle. If *`mask`* is attached to an Element at the time **wfcDestroyMask** is called, the Mask continues to exist until it is detached or the Element is destroyed.

ERRORS

`WFC_ERROR_BAD_HANDLE`
- if *`mask`* is not a valid Mask associated with *`dev`*

7 Composition Elements

Composition Elements are the fundamental unit used to specify how content should be composed. A scene consists of zero or more Elements stacked over a background plane. Logically, each Element exists in a distinct plane parallel to the background plane, between the background and the viewer. There is no concept of distance in the z dimension between Elements, only ordering.

Composition is equivalent to blending each Element on top of the destination buffer according to the relative ordering of the Elements. The result of composition is a 2D image.

An Element specifies an integer rectangular area of the destination within which it will generate new color values for each pixel.

If the Element makes use of any transparency, the final destination color values will be the result of blending color data defined solely by the Element together with existing color data retrieved from the destination image. Transparency and blending is described in Section 7.1.7.

The pixel content of an Element is determined by a 2D source image, a `WFCSource`, that is attached to the Element. The source image encoding does not need to match the destination image encoding in terms of pixel format, alpha convention or color space. Provision of `WFCSource`s is described in Section 6.1.

The Element specifies a source rectangle within its source image. The content of the source rectangle is scaled to fill the destination rectangle, which allows different scaling in the x and y dimensions. Filtering may be applied and is described in Section 7.1.6.

The mapping of a source rectangle to a destination area can be subject to both rotation and flipping. Rotation is limited to 90-degree increments. Flipping is performed about the vertical axis running through the center of the source rectangle and is defined to occur before rotation.

7.1 Element Attributes

An Element consists of attributes enumerated by `WFCElementAttrib`.

```
typedef enum {
    WFC_ELEMENT_DESTINATION_RECTANGLE       = 0x7101,
    WFC_ELEMENT_SOURCE                      = 0x7102,
    WFC_ELEMENT_SOURCE_RECTANGLE            = 0x7103,
    WFC_ELEMENT_SOURCE_FLIP                 = 0x7104,
    WFC_ELEMENT_SOURCE_ROTATION             = 0x7105,
    WFC_ELEMENT_SOURCE_SCALE_FILTER         = 0x7106,
    WFC_ELEMENT_TRANSPARENCY_TYPES          = 0x7107,
    WFC_ELEMENT_GLOBAL_ALPHA                = 0x7108,
    WFC_ELEMENT_MASK                        = 0x7109
} WFCElementAttrib;
```

Element attributes along with their default values are specified in ***Table 4***.

Attribute	Type	R/W	Default
WFC_ELEMENT_DESTINATION_ RECTANGLE	WFCint[4]	R/W	(0, 0, 0, 0)
WFC_ELEMENT_SOURCE	WFCSource	R/W	WFC_INVALID_HANDLE
WFC_ELEMENT_SOURCE_RECTANGLE	WFCfloat[4]	R/W	(0, 0, 0, 0)
WFC_ELEMENT_SOURCE_FLIP	WFCboolean	R/W	WFC_FALSE
WFC_ELEMENT_SOURCE_ROTATION	WFCRotation	R/W	WFC_ROTATION_0
WFC_ELEMENT_SOURCE_SCALE_FILTER	WFCScaleFilter	R/W	WFC_SCALE_ FILTERING_NONE
WFC_ELEMENT_TRANSPARENCY_TYPES	WFCbitfield	R/W	0
WFC_ELEMENT_GLOBAL_ALPHA	WFCfloat	R/W	1
WFC_ELEMENT_MASK	WFCMask	R/W	WFC_INVALID_HANDLE

Table 4 – Default Element attributes

7.1.1 Destination Rectangle

`WFC_ELEMENT_DESTINATION_RECTANGLE` defines the placement of the Element's content within the Context coordinate system.

The destination rectangle may be partly or wholly outside of the bounds of the Element's Context, i.e. the bounds defined by the Context attributes `WFC_CONTEXT_TARGET_{HEIGHT|WIDTH}`. Element rendering will be clipped to the intersection of the destination rectangle and the Context bounds.

This attribute is a four integer array in the form (*`offsetX, offsetY, width, height`*). The top-left pixel of this Element will be positioned at (*`offsetX, offsetY`*). *`width`* and *`height`* must not be negative. If either *`width`* or *`height`* is zero, this Element will not affect composition results.

Accessors: **wfcGetElementAttribiv, wfcSetElementAttribiv, wfcGetElementAttribfv, wfcSetElementAttribfv**

7.1.2 Source Image

`WFC_ELEMENT_SOURCE` defines the Element's Source image provider that will supply the color data. This value must be `WFC_INVALID_HANDLE` or a valid `WFCSource` associated with the same Context as the Element. If this value is `WFC_INVALID_HANDLE`, this Element will not affect composition results.
It is legitimate for multiple Elements to use the same Source.

Setting this attribute to a valid Source causes a reference to be placed on the Source by this Element. Updating this attribute removes any reference that the Element previously placed on a Source.

Accessors: **wfcGetElementAttribi, wfcSetElementAttribi**

7.1.3 Source Rectangle

`WFC_ELEMENT_SOURCE_RECTANGLE` defines the rectangular sub-area of the Element's Source that will be used when rendering the Element. It is specified in the Source image coordinate space.

This attribute is a four float array in the form (*`offsetX`*, *`offsetY`*, *`width`*, *`height`*). The top-left of the source rectangle is located at (*`offsetX`*, *`offsetY`*). All values must not be negative. If either *`width`* or *`height`* is zero, this Element will not affect composition results.

When **wfcCommit** is called, the source rectangle must be fully contained inside the Source for the scene to be considered consistent. As a consequence, (*`offsetX`* + *`width`*) and (*`offsetY`* + *`height`*) must be no greater than the Source width and height, respectively. [13]

Accessors: **wfcGetElementAttribiv, wfcSetElementAttribiv, wfcGetElementAttribfv, wfcSetElementAttribfv**

[13] When using fractional values for the width or height of a source rectangle that approaches the right or bottom edges of the source image, it is important for the user to verify that the resultant bottom right coordinate is in fact within the bounds of the source. Inaccuracies in floating-point calculations can lead to non-intuitive results where the source rectangle fractionally exceeds the source bounds. It is the user's responsibility to check for and resolve this situation. One solution is to subtract a small amount (e.g. 0.00001) from the width or height.

7.1.4 Source Flipping

`WFC_ELEMENT_SOURCE_FLIP` determines whether the flip feature is enabled. Flipping consists of inverting the Crop stage output image top-to-bottom about the horizontal centerline of the image.

Accessors: **wfcGetElementAttribi, wfcSetElementAttribi**

7.1.5 Source Rotation

`WFC_ELEMENT_SOURCE_ROTATION` defines a clockwise rotation to be applied to the contents of Flipping stage output. The value must be a valid `WFCRotation`.

Accessors: **wfcGetElementAttribi, wfcSetElementAttribi**

7.1.6 Source Scaling Filter

`WFC_ELEMENT_SOURCE_SCALE_FILTER` determines the type of filtering that can be used when scaling content. Filtering involves determining an approximate value for a source point using a function of the nearby pixel center values.

The `WFCScaleFilter` enumeration defines the only legal values for this attribute.

```
typedef enum {
    WFC_SCALE_FILTER_NONE   = 0x7151,  /* default */
    WFC_SCALE_FILTER_FASTER = 0x7152,
    WFC_SCALE_FILTER_BETTER = 0x7153
} WFCScaleFilter;
```

If *`filtering`* is `WFC_SCALE_FILTER_NONE`, filtering is disabled. Elements are drawn using point sampling (also known as nearest-neighbor replication) only.

If *`filtering`* is `WFC_SCALE_FILTER_FASTER`, low-to-medium quality filtering that does not require extensive additional resource allocation is used.

If *`filtering`* is `WFC_SCALE_FILTER_BETTER`, high-quality filtering that may allocate additional memory for pre-filtering, tables, and the like is used.

Implementations are not required to provide three distinct filtering algorithms, but the point sampling mode must be supported.

If the point sampling mode is used and the source-destination mapping results in a 1:1 mapping between source pixels and destination pixels, each destination pixel value must be unaffected by any neighboring pixel values surrounding the corresponding source pixel.

Filtering must be continuous across the source image. The result of filtering a particular source point must be determined solely by the contents of the source image and the selected filtering mode[14]. The implementation is permitted to access pixels within the source image that lie outside of the source rectangle. If pixels that lie outside the source image are required, the implementation must generate these non-existent pixels from the source image using a consistent scheme, ideally edge replication.

Accessors: **wfcGetElementAttribi, wfcSetElementAttribi**

7.1.7 Transparency Types

`WFC_ELEMENT_TRANSPARENCY_TYPES` determines the active transparency features. Multiple transparency types can be enabled through ORing individual types.

The `WFCTransparencyType` enumeration defines values for each transparency type.

```
typedef enum {
    WFC_TRANSPARENCY_NONE                  = 0,
    WFC_TRANSPARENCY_ELEMENT_GLOBAL_ALPHA = (1 << 0),
    WFC_TRANSPARENCY_SOURCE                = (1 << 1),
    WFC_TRANSPARENCY_MASK                  = (1 << 2)
} WFCTransparencyType;
```

If this attribute is equal to `WFC_TRANSPARENCY_NONE`, the Element will be considered to be opaque. The generated pixel values within this Element's destination rectangle will be unaffected by Elements beneath this one.

If `WFC_TRANSPARENCY_ELEMENT_GLOBAL_ALPHA` is enabled, this Element's global alpha (specified by `WFC_ELEMENT_GLOBAL_ALPHA`) will be used when blending this Element's content with the destination.

If `WFC_TRANSPARENCY_SOURCE` is enabled, any alpha channel inherent in this Element's Source (specified by `WFC_ELEMENT_SOURCE`) will be considered when filtering and when blending this Element's content with the destination.

[14] This implies that the source rectangle associated with the Element being rendered does not affect the pixels that are read by the filtering kernel.

Otherwise the alpha channel of the Source will not affect the rendering of the Element.

If `WFC_TRANSPARENCY_MASK` is enabled, this Element's Mask (specified by `WFC_ELEMENT_MASK`) will be used when blending this Element's content with the destination.

The use of multiple transparency types is supported for the following combinations only:

- `WFC_TRANSPARENCY_ELEMENT_GLOBAL_ALPHA | WFC_TRANSPARENCY_SOURCE`
- `WFC_TRANSPARENCY_ELEMENT_GLOBAL_ALPHA | WFC_TRANSPARENCY_MASK`

Blending Functions

The effect of using transparency is defined in terms of an alpha blending function $a(a_{src}, a_{dst})$ and a color blending function $c'(c'_{src}, c'_{dst}, a_{src}, a_{dst})$ where $c' = \alpha * c$ is a premultiplied color. Given a premultiplied color and alpha source tuple $(R_{src}, G_{src}, B_{src}, a_{src})$ and a premultiplied color and alpha destination tuple $(R_{dst}, G_{dst}, B_{dst}, a_{dst})$, blending produces the blended premultiplied color and alpha tuple $(c'(R_{src}, R_{dst}, a_{src}, a_{dst}), c'(G_{src}, G_{dst}, a_{src}, a_{dst}), c'(B_{src}, B_{dst}, a_{src}, a_{dst}), a(a_{src}, a_{dst}))$. This value is converted to the destination format and replaces the destination.

The source and destination are converted to premultiplied alpha format prior to blending. If either the source or destination format does not contain an alpha channel, an alpha value of 1 is used in place of a_{src} or a_{dst} accordingly. If `WFC_TRANSPARENCY_SOURCE` is disabled, the conversion of the source to premultiplied alpha format ignores the source's alpha values and instead uses an alpha value of 1.

In the blending functions below, the following notation is used:

- c'_{src} and a_{src} represent the premultiplied source color and alpha values
- c'_{dst} and a_{dst} represent the premultiplied destination color and alpha values
- a_{ele} represents the Element global alpha value
- a_{mask} represents the alpha value from the Mask

The blending functions for each transparency mode are as follows.

`WFC_TRANSPARENCY_NONE`
$c'(c'_{src}, c'_{dst}, a_{src}, a_{dst}) = c'_{src}$

$a(a_{src}, a_{dst}) = 1$

WFC_TRANSPARENCY_ELEMENT_GLOBAL_ALPHA

$c'(c'_{src}, c'_{dst}, a_{src}, a_{dst}) = c'_{src} * a_{ele} + c'_{dst} * (1 - a_{ele})$

$a(a_{src}, a_{dst}) = a_{ele} + a_{dst} * (1 - a_{ele})$

WFC_TRANSPARENCY_SOURCE

$c'(c'_{src}, c'_{dst}, a_{src}, a_{dst}) = c'_{src} + c'_{dst} * (1 - a_{src})$

$a(a_{src}, a_{dst}) = a_{src} + a_{dst} * (1 - a_{src})$

WFC_TRANSPARENCY_MASK

$c'(c'_{src}, c'_{dst}, a_{src}, a_{dst}) = c'_{src} * a_{mask} + c'_{dst} * (1 - a_{mask})$

$a(a_{src}, a_{dst}) = a_{mask} + a_{dst} * (1 - a_{mask})$

WFC_TRANSPARENCY_ELEMENT_GLOBAL_ALPHA | WFC_TRANSPARENCY_SOURCE

$c'(c'_{src}, c'_{dst}, a_{src}, a_{dst}) = c'_{src} * a_{ele} + c'_{dst} * (1 - a_{src} * a_{ele})$

$a(a_{src}, a_{dst}) = a_{src} * a_{ele} + a_{dst} * (1 - a_{src} * a_{ele})$

WFC_TRANSPARENCY_ELEMENT_GLOBAL_ALPHA | WFC_TRANSPARENCY_MASK

$c'(c'_{src}, c'_{dst}, a_{src}, a_{dst}) = c'_{src} * a_{mask} * a_{ele} + c'_{dst} * (1 - a_{mask} * a_{ele})$

$a(a_{src}, a_{dst}) = a_{mask} * a_{ele} + a_{dst} * (1 - a_{mask} * a_{ele})$

Note that for on-screen composition, the destination alpha value is always 1 because on-screen Contexts are defined to have an opaque background color.

Accessors: **wfcGetElementAttribi, wfcSetElementAttribi**

7.1.8 Global Alpha

`WFC_ELEMENT_GLOBAL_ALPHA` determines a single opacity value that affects the blending of this Element's content, if Element global alpha is enabled via `WFC_ELEMENT_TRANSPARENCY_TYPES`.

When operated on using float accessors, this attribute's value ranges between 0 and 1 where 0 represents fully transparent and 1 represents fully opaque. Attempting to set values outside this range will fail.

When operated on using integer accessors, this attribute's value ranges between 0 and 255 where 0 represents fully transparent and 255 represents fully opaque. Attempting to set values outside this range will fail. The attribute is converted to an integer using `round(255.0*floatValue)` when an integer query is used. If an integer setter is used, the incoming value is divided by 255.0f to obtain a floating-point value between 0 and 1.

If Element global alpha is enabled and this value is zero, this Element will not affect composition results.

Accessors: **wfcGetElementAttribi, wfcSetElementAttribi, wfcGetElementAttribf, wfcSetElementAttribf**

7.1.9 Mask

`WFC_ELEMENT_MASK` defines a Mask image provider of per-pixel alpha values that affects the blending of this Element's content, if Mask alpha is enabled via `WFC_ELEMENT_TRANSPARENCY_TYPES`. This value must be `WFC_INVALID_HANDLE` or a valid Mask associated with the same Context as the Element. If this value is `WFC_INVALID_HANDLE`, no masking will take place. It is equivalent to attaching a mask with an alpha value of 1 in every pixel. It is legitimate for multiple Elements to use the same Mask.

Setting this attribute to a valid Mask causes a reference to be placed on the Mask by this Element. Updating this attribute removes any reference that the Element previously placed on a Mask.

If a Mask is set, it must be equal in size to the Element's destination rectangle specified by `WFC_ELEMENT_DESTINATION_RECTANGLE` when **wfcCommit** is called. Otherwise the scene will be considered inconsistent. This restriction may be lifted in future revisions of this specification.

Accessors: **wfcGetElementAttribi, wfcSetElementAttribi**

7.2 Creating Elements

An Element is created for use with a specific context by calling **wfcCreateElement.**

```
WFCElement wfcCreateElement(WFCDevice     dev,
                            WFCContext    ctx
                            const WFCint *attribList);
```

If **wfcCreateElement** succeeds, a new `WFCElement` is initialized with default attributes and a handle to it is returned. The element is specific to *`ctx`* and can not be used with another Context.

The structure of *`attribList`* is described in Section 2.9. No valid attributes are defined in this specification.

On failure, **wfcCreateElement** returns `WFC_INVALID_HANDLE`.

ERRORS

`WFC_ERROR_OUT_OF_MEMORY`
- if the implementation fails to allocate resources for the Element

`WFC_ERROR_BAD_ATTRIBUTE`
- if *`attribList`* contains an invalid attribute

`WFC_ERROR_BAD_HANDLE`
- if *`ctx`* is not a valid Context associated with *`dev`*

7.3 Querying Element Attributes

Element attributes can be queried by calling:

```
WFCint wfcGetElementAttribi(  WFCDevice        dev,
                              WFCElement       element,
                              WFCElementAttrib attrib);

WFCfloat wfcGetElementAttribf(WFCDevice        dev,
                              WFCElement       element,
                              WFCElementAttrib attrib);

void wfcGetElementAttribiv(   WFCDevice        dev,
                              WFCElement       element,
                              WFCElementAttrib attrib,
                              WFCint           count,
                              WFCint          *values);

void wfcGetElementAttribfv(   WFCDevice        dev,
                              WFCElement       element,
                              WFCElementAttrib attrib,
                              WFCint           count,
                              WFCfloat        *values);
```

The *dev* and *element* parameters denote the specific Device and Element being queried. The *attrib* parameter denotes the attribute to retrieve and return via the function return value or the value parameter. For the vector-based functions, *count* denotes the number of values to retrieve and *values* must be an array of *count* elements.

On success, the value(s) of *attrib* for *element* is returned via the function return value or the *values* parameter. On failure, *values* in not modified. Refer to Section 7.1 for a list of valid attributes.

Retrieving handle-typed attributes does not affect the lifetime of the handle.

ERRORS

WFC_ERROR_BAD_ATTRIBUTE
- if *attrib* is an invalid attribute
- if *attrib* does not permit the use of this accessor

WFC_ERROR_ILLEGAL_ARGUMENT
- if *count* does not match the size of *attrib*
- if *values* is NULL or not aligned with its datatype

`WFC_ERROR_BAD_HANDLE`
- if *`element`* is not a valid Element associated with *`dev`*

7.4 Setting Element Attributes

Element attributes can be set using:

```
void wfcSetElementAttribi( WFCDevice        dev,
                           WFCElement       element,
                           WFCElementAttrib attrib
                           WFCint           value);

void wfcSetElementAttribf( WFCDevice        dev,
                           WFCElement       element,
                           WFCElementAttrib attrib
                           WFCfloat         value);

void wfcSetElementAttribiv(WFCDevice        dev,
                           WFCElement       element,
                           WFCElementAttrib attrib,
                           WFCint           count,
                           const WFCint    *values);

void wfcSetElementAttribfv(WFCDevice        dev,
                           WFCElement       element,
                           WFCElementAttrib attrib,
                           WFCint           count,
                           const WFCfloat  *values);
```

The *`dev`* and *`element`* parameters denote the specific Device and Element, respectively, associated with this function call. The *`attrib`* parameter denotes the attribute being set. For the vector-based functions, *`count`* denotes the number of values provided and *`values`* must be an array of *`count`* elements.

On success, the value(s) of *`attrib`* for *`element`* are set to value. On failure, the Element is not modified. Refer to Section 7.1 for a list of valid attributes.

Note that **wfcCommit** must be called on the associated Context before updated values will affect rendering.

ERRORS

`WFC_ERROR_BAD_ATTRIBUTE`
- if *`attrib`* is an invalid attribute
- if *`attrib`* is an immutable attribute
- if *`attrib`* does not permit the use of this accessor

WFC_ERROR_ILLEGAL_ARGUMENT
- if *count* does not match the size of *attrib*
- if *values* is NULL
- if *values* is non-NULL and not aligned with its datatype
- if the user attempts to set a valid attribute to an invalid value

WFC_ERROR_BAD_HANDLE
- if *element* is not a valid Element associated with *dev*

7.5 Ordering Elements

For an Element to participate in composition, it must be inside its Context's scene. The scene is an ordered list of elements that is used for rendering. Upon creation, Elements are outside of their Context's scene, meaning they do not affect the results of composition. Modifications to Element ordering are observable immediately through query functions such as **wfcGetElementAbove** but are not realized until **wfcCommit** is called on the associated Context.

7.5.1 Inserting Elements

To place an Element into its Context's scene, call **wfcInsertElement.**

```
void wfcInsertElement(WFCDevice  dev,
                      WFCElement element,
                      WFCElement subordinate);
```

element is placed inside its Context's scene immediately above *subordinate*. Any Element that was immediately above *subordinate* is placed immediately above *element*. If *element* is already inside the scene, it is effectively removed and re-inserted in the new location. Inserting an Element above itself is permitted but has no effect.

If *subordinate* is WFC_INVALID_HANDLE, the element is placed at the bottom of the scene. Otherwise, *element* and *subordinate* must have been created using the same context and *subordinate* must be in the scene.

ERRORS

WFC_ERROR_ILLEGAL_ARGUMENT
- if *subordinate* and *element* were not created using the same context
- if *subordinate* is outside the scene

WFC_ERROR_BAD_HANDLE
- if *element* is not a valid Element associated with *dev*
- if *subordinate* is not WFC_INVALID_HANDLE and is not a valid Element associated with *dev*

7.5.2 Removing Elements

To remove an Element from its Context's scene, call **wfcRemoveElement.**

```
void wfcRemoveElement(WFCDevice  dev,
                      WFCElement element);
```

element is placed outside the scene and will not affect composition. Removing an Element that is already outside the scene has no effect.

ERRORS

WFC_ERROR_BAD_HANDLE
- if *element* is not a valid Element associated with *dev*

7.5.3 Querying Element Ordering

To determine the Element above a specified Element call **wfcGetElementAbove**.

```
WFCElement wfcGetElementAbove(WFCDevice  dev,
                              WFCElement element);
```

element must be inside its Context's scene. If *element* is top-most or outside its Context's scene, WFC_INVALID_HANDLE is returned. Otherwise the Element above *element* is returned. This call does not affect the lifetime or validity of the Element returned.

ERRORS

WFC_ERROR_ILLEGAL_ARGUMENT
- if *element* is outside the scene

WFC_ERROR_BAD_HANDLE
- if *element* is not a valid Element associated with *dev*

To determine the Element below a specified Element call **wfcGetElementBelow**.

```
WFCElement wfcGetElementBelow(WFCDevice  dev,
                              WFCElement element);
```

element must be inside its Context's scene. If *element* is bottom-most or outside its Context's scene, WFC_INVALID_HANDLE is returned. Otherwise the Element below *element* is returned. This call does not affect the lifetime or validity of the Element returned.

ERRORS

`WFC_ERROR_ILLEGAL_ARGUMENT`
- if *`element`* is outside the scene

`WFC_ERROR_BAD_HANDLE`
- if *`element`* is not a valid Element associated with *`dev`*

7.6 Destroying Elements

To destroy an Element, call **wfcDestroyElement.**

```
void wfcDestroyElement(WFCDevice  dev,
                       WFCElement element);
```

All resources associated with *`element`* are marked for deletion as soon as possible. Any references held by *`element`* on Sources or Masks are removed when this Element is deleted. Following the call, *`element`* is no longer a valid Element handle. If *`element`* is inside its Context's scene at the time **wfcDestroyElement** is called, it is removed from the scene. This removal does not affect the Context's committed scene until **wfcCommit** is called.

ERRORS

`WFC_ERROR_BAD_HANDLE`
- if *`element`* is not a valid Element associated with *`dev`*

8 Rendering

OpenWF Composition is designed to support both *autonomous* and *user-driven* composition modes. These modes correspond to a Context being *active* or *inactive* respectively.

User-driven Composition

Upon creation Contexts are inactive. In this state, rendering is user-driven meaning that the rendering of each frame must be requested by the user via a distinct call to **wfcCompose**.

If the user allows previous composition requests to complete and does not make any further requests, they can be sure that Composition will not access the render target or any image providers. This allows the user to update the contents of single-buffered image providers atomically with respect to Composition.

Autonomous Composition

If a Context is active, rendering is performed autonomously without user intervention. The most recently committed scene is used repeatedly to render the latest content from the referenced image providers until the user commits a new scene or deactivates the Context.

The implementation decides when to render. Potential scheduling implementations include constant periodic composition as well as demand-driven composition. Scene updates (achieved via **wfcCommit**) and content updates from image providers do not necessarily result in immediate rendering. Artificial delays may be introduced in order to increase overall system efficiency. Implementations that allow scene updates to occur more frequently than composition rendering occurs should drop scene updates that are superseded before being rendered.

To allow the display of content that is sensitive to audio-visual synchronization, the implementation must provide some mechanism for allowing frames to be displayed at a specific time, subject to a degree of tolerance. For example, the implementation may need to provide feedback to the native stream on the historical delay incurred when displaying previous frames. The tolerances and mechanisms for achieving this synchronization are not specified by OpenWF Composition.

Whilst a Context has autonomous composition enabled, Composition may constantly access the image providers referenced in the committed scene. The user can stop Composition from accessing a particular image provider by either,

- committing a new scene that does not reference the particular image provider, or
- disabling autonomous composition on the Context

8.1 Activation

A Context is activated by calling **wfcActivate**.

```
void wfcActivate(WFCDevice  dev,
                 WFCContext ctx);
```

If successful, *ctx* will begin performing autonomous composition using *ctx*'s committed scene, as defined in the previous call to **wfcCommit** on *ctx*. The target of *ctx* and all image providers associated with the *ctx*'s committed scene will be considered in use by *ctx*. If *ctx* already has autonomous composition enabled, this function has no effect.

If the target of *ctx* is already in use as a render target by another renderer, the target's contents become undefined and *ctx* must be deactivated before the user can resume defined rendering to the target.

Note that completion of this function means only that *ctx* is permitted to begin rendering and does not imply that any rendering has started or finished.

ERRORS

WFC_ERROR_BAD_HANDLE
- if *ctx* is not a valid Context associated with *dev*

8.2 Deactivation

A context is deactivated by calling **wfcDeactivate**.

```
void wfcDeactivate(WFCDevice  dev,
                   WFCContext ctx);
```

ctx will asynchronously stop performing autonomous composition. Any composition in progress will be allowed to complete normally. If *ctx* already has autonomous composition disabled, this function has no effect.

wfcDeactivate may return before *ctx* has finished rendering. Completion of this request, as can be determined by synchronization functions (see Section 9),

implies that any previous calls to **wfcCommit** on *ctx* have completed and that *ctx* is no longer accessing any target images or image providers.[15]

If the target of *ctx* is capable of being targeted by multiple renderers, completion also implies that the target is now available for those other renderers to use as a render target.

ERRORS

WFC_ERROR_BAD_HANDLE
- if *ctx* is not a valid Context associated with *dev*

8.3 Composition Requests

An inactive Context can be requested to compose by calling **wfcCompose**.

```
void wfcCompose(WFCDevice  dev,
                WFCContext ctx,
                WFCboolean wait);
```

If **wfcCompose** succeeds, *ctx*'s committed scene, as defined in the previous call to **wfcCommit** on *ctx*, will be asynchronously rendered to the Context's destination. The target of *ctx* and all image providers associated with the *ctx*'s committed scene will be considered in use by *ctx* during this time. This rendering will not be affected by any subsequent changes to *ctx*'s committed scene.

Composition is equivalent to clearing the destination to the background color, erasing any previous pixel values, followed by iterating over the Elements inside the scene, rendering each element on top of the previous Elements.

ctx must be inactive. If the target of ctx is already in use as a render target by another renderer, the target's contents become undefined.

If *wait* is WFC_FALSE and *ctx* is still processing a previous call to **wfcCompose**, this call will fail immediately without causing any rendering. The previous request will not be affected. If *wait* is WFC_TRUE, **wfcCompose** will not fail due to a previous request being incomplete[16]. This may entail a delay in

[15] For on-screen composition, this implies that the screen contents must be cached so that the contents continue to be displayed without further access to the source images. Such caching may be trivial if the physical display naturally persists its contents.

[16] The two types of behaviour aid user flexibility. Users may choose "immediate rejection" if they are controlling multiple contexts from a single thread and do not wish to be stalled for a long period. Users responsible for a single context may benefit from "stall until ready" rather than being forced to poll.

the call returning, due to the implementation waiting for a previous request to complete.

As a consequence of rendering being asynchronous, all image providers and any target image associated with the composition request may continue to be accessed by the implementation after **wfcCompose** returns. If *ctx*'s target is multi-buffered, previously rendered images may be accessed for the purpose of optimizing rendering[17].

There is a finite period of time between the call to **wfcCompose** returning and the completion of the request. Synchronization functions, described in Section 9, must be used if the user needs to know if or when a particular call to **wfcCompose** has completed. Such functions enable the user to monitor the time taken to process requests.

Completion of this request implies that *ctx* is ready to accept a new request and that all rendering associated with the request has finished; any target image and all image providers associated with the request are no longer in use by Composition. If the target of *ctx* is also capable of being targeted by multiple renderers, completion also implies that the target is now available for those other renderers to use as a render target.

ERRORS

`WFC_ERROR_UNSUPPORTED`
- if *ctx* is active

`WFC_ERROR_BUSY`
- if *wait* is `WFC_FALSE` and *ctx* can not immediately accept any more composition requests

`WFC_ERROR_BAD_HANDLE`
- if *ctx* is not a valid Context associated with *dev*

[17] For destination regions that require rendering but are up to date within other target images, this allows the implementation to fetch destination pixel values from previously composed target images rather than recalculating them from source images.

9 Synchronization

The Composition system supports the use of reusable EGL Sync Objects for synchronization. Sync objects may be used for synchronization of operations between the Composition system and other EGL client APIs, and for synchronizing between multiple Composition Contexts, among other purposes. Refer to [EGL08] for details on an EGL Sync Objects.

Sync objects have a status value with two possible states: signaled and unsignaled. Events representing completion of a Context's operations may be associated with a reusable sync object. When an event is initially associated with a sync object, the object is unsignaled (its status is set to unsignaled). When an associated event occurs, the object is signaled (its status is set to signaled).

```
void wfcFence(WFCDevice     dev,
              WFCContext    ctx,
              WFCEGLDisplay dpy,
              WFCEGLSync    sync);
```

On success, **wfcFence** inserts a fence in *ctx*'s command stream and associates it with the specified sync object *sync*. Calling **wfcFence** unsignals *sync*.

dpy must be a valid `EGLDisplay` object cast into the type `WFCEGLDisplay`. *sync* must be a valid `EGLSyncKHR` object cast into the type `WFCEGLSync`. *sync* must have an `EGL_SYNC_TYPE_KHR` attribute equal to `EGL_SYNC_REUSABLE_KHR`. *sync* must have been created for *dpy*. If *dpy* does not match the `EGLDisplay` used to create *sync*, the behavior is undefined.

When all commands previously issued on *ctx* complete, *sync* is signaled by the Composition system, causing any commands blocking on *sync* to unblock, e.g. **eglClientWaitSyncKHR**. No other state is affected by execution of the fence command.

It is legitimate to reuse a sync object by issuing a further fence command on a sync object after a previous fence command has completed and signaled the same sync object. However the user is not allowed to issue more than one outstanding fence command on a given sync object at a time. If a fence command is issued on a sync object that already has a pending fence command, the behavior is undefined.

ERRORS

`WFC_ERROR_ILLEGAL_ARGUMENT`
- if *`dpy`* is not a valid `EGLDisplay`
- if *`sync`* is not a valid sync object
- if *`sync`*'s `EGL_SYNC_TYPE_KHR` is not `EGL_SYNC_REUSABLE_KHR`

`WFC_ERROR_BAD_HANDLE`
- if *`ctx`* is not a valid Context associated with *`dev`*

10 Extending Composition

OpenWF Composition is designed to be extended. An extension may define new datatypes, new values for existing parameter types and new functions. An extension must have no effect on programs that do not enable any of its features.

10.1 Extension Naming Conventions

An OpenWF Composition extension is named by a string of the form `WFC_`*`type`*`_`*`name`*, where *`type`* is either the string `EXT` or a vendor-specific string and *`name`* is a name assigned by the extension author. A letter `X` added to the end of type indicates that the extension is experimental.

Values (e.g. enumerated values or preprocessor `#define`s) defined by an extension carry the suffix `_`*`type`*. Functions and datatypes carry the suffix *`type`* without a separating underscore.

The `<WF/wfcext.h>` header file defines the values, functions and datatypes that may be available on a platform. The file will define a preprocessor macro with the name `WFC_`*`type`*`_`*`name`* and a value of `1` for each supported extension.

10.2 The Extension Registry

Khronos, or its designee, will maintain a publicly-accessible registry of extensions. This registry will contain, for each extension, at least the following information:

- The name of the extension in the form `WFC_`*`type`*`_`*`name`*
- An email address of a contact person
- A list of dependencies on other extensions
- A statement on the IP status of the extension
- An overview of the scope and semantics of the extension
- New functions defined by the extension
- New datatypes defined by the extension
- New values to be added to existing enumerated datatypes
- Additions and changes to the OpenWF Composition specification
- New errors generated by functions affected by the extension
- New state defined by the extension
- Authorship information and revision history

10.3 Using Extensions

Extensions may be detected statically, by means of preprocessor symbols, or dynamically, by means of the **wfcGetStrings** or **wfcIsExtensionSupported** functions. Using static detection of extensions at compile time is generally not

sufficient to ensure an implementation supports a particular extension. When static detection is used, a runtime dynamic check for the extension should also be made to ensure proper support.

Extension functions may be included in application code statically by placing appropriate `#ifdef` directives around functions that require the presence of a particular extension, and may also be accessed dynamically through function pointers returned by **eglGetProcAddress** or by other platform-specific means.

10.3.1 Accessing Extensions Statically

The extensions defined by a given platform are defined in the `<WF/wfcext.h>` header file, or in header files automatically included by `<WF/wfcext.h>`. In order to write applications that run on platforms with and without a given extension, conditional compilation based on the presence of the extension's preprocessor macro may be used.

```
#ifdef WFC_EXT_my_extension
    wfcMyExtensionFuncEXT(...);
#endif
```

10.3.2 Accessing Extensions Dynamically

OpenWF Composition contains a mechanism for applications to access information about the runtime platform, and to access extensions that may not have been present when the application was compiled.

The `WFCStringID` enumeration defines values for strings that the user can query.

```
typedef enum {
    WFC_VENDOR     = 0x7200,
    WFC_RENDERER   = 0x7201,
    WFC_VERSION    = 0x7202,
    WFC_EXTENSIONS = 0x7203
} WFCStringID;
```

The **wfcGetStrings** function returns information about the OpenWF Composition implementation, including extension information.

```
WFCint wfcGetStrings(WFCDevice        dev,
                     WFCStringID      name,
                     const char     **strings,
                     WFCint           stringsCount);
```

The user provides a buffer to receive a list of pointers to strings specific to *dev*. *strings* is the address of the buffer. *stringsCount* is the number of string pointers that can fit into the buffer. If *dev* or *name* is not valid, zero is returned and the buffer is not modified.

If *strings* is NULL, the total number of *name*-related strings is returned and the buffer is not modified.

If *strings* is not NULL, the buffer is populated with a list of *name*-related string pointers. The strings are read-only and owned by the implementation. No more than *stringsCount* pointers will be written even if more are available. The number of string pointers written into *strings* is returned. If *stringsCount* is negative, no string pointers will be written.

The combination of WFC_VENDOR and WFC_RENDERER may be used together as a platform identifier by applications that wish to recognize a particular platform and adjust their algorithms based on prior knowledge of platform bugs and performance characteristics.

If *name* is WFC_VENDOR, a single string of the name of company responsible for this OpenWF Composition implementation is returned.

If *name* is WFC_RENDERER, a single string of the name of the renderer is returned. This name is typically specific to a particular configuration of a hardware platform, and does not change from release to release.

If *name* is WFC_VERSION, a single string of the version number of the specification implemented by the renderer is returned as a string in the form *major_number.minor_number*. For this specification, "1.0" is returned.

If *name* is WFC_EXTENSIONS, a list of strings denoting the supported extensions to OpenWF Composition is returned.

ERRORS

WFC_ERROR_ILLEGAL_ARGUMENT
- if *stringsCount* is negative
- if *name* is not a valid string ID

The **wfcIsExtensionSupported** function provides an alternate means of testing for support of a specific extension.

```
WFCboolean wfcIsExtensionSupported(WFCDevice       dev,
                                   const char     *string);
```

`WFC_TRUE` will be returned if the extension denoted by *`string`* is supported by *`dev`*. Otherwise `WFC_FALSE` will be returned.

Functions defined by an extension may be accessed by means of a function pointer obtained from the EGL function **eglGetProcAddress**.

10.4 Creating Extensions

Any vendor may define a vendor-specific extension. Each vendor should apply to Khronos to obtain a vendor string and any numerical token values required by the extension.

An OpenWF Composition extension may be deemed a shared extension if two or more vendors agree in good faith to ship an extension, or the Khronos OpenWF Composition working group determines that it is in the best interest of its members that the extension be shared. A shared extension may be adopted (with appropriate naming changes) into a subsequent release of the OpenWF Composition specification.

11 Appendix A: Header Files

This section defines a minimal C language header file for the type definitions and functions of OpenWF Composition. The actual header file provided by a platform vendor may differ from the one shown here.

wfc.h

```
/*------------------------------------------------------------------------*
 *                                                                        *
 * Sample implementation of wfc.h, version 1.0, draft 22                  *
 *                                                                        *
 * Copyright © 2007-2009 The Khronos Group                                *
 *                                                                        *
 *------------------------------------------------------------------------*/

#ifndef _WFC_H_
#define _WFC_H_

#include <WF/wfcplatform.h>

#ifdef __cplusplus
extern "C" {
#endif

#define OPENWFC_VERSION_1_0       (1)

#define WFC_NONE                  (0)

#define WFC_INVALID_HANDLE        ((WFCHandle)0)

#define WFC_DEFAULT_DEVICE_ID     (0)

#define WFC_MAX_INT               ((WFCint)16777216)
#define WFC_MAX_FLOAT             ((WFCfloat)16777216)

typedef WFCHandle WFCDevice;
typedef WFCHandle WFCContext;
typedef WFCHandle WFCSource;
typedef WFCHandle WFCMask;
typedef WFCHandle WFCElement;

typedef enum {
    WFC_ERROR_NONE                          = 0,
    WFC_ERROR_OUT_OF_MEMORY                 = 0x7001,
    WFC_ERROR_ILLEGAL_ARGUMENT              = 0x7002,
    WFC_ERROR_UNSUPPORTED                   = 0x7003,
    WFC_ERROR_BAD_ATTRIBUTE                 = 0x7004,
    WFC_ERROR_IN_USE                        = 0x7005,
    WFC_ERROR_BUSY                          = 0x7006,
    WFC_ERROR_BAD_DEVICE                    = 0x7007,
    WFC_ERROR_BAD_HANDLE                    = 0x7008,
    WFC_ERROR_INCONSISTENCY                 = 0x7009,
    WFC_ERROR_FORCE_32BIT                   = 0x7FFFFFFF
```

```
} WFCErrorCode;

typedef enum {
    WFC_DEVICE_FILTER_SCREEN_NUMBER         = 0x7020,
    WFC_DEVICE_FILTER_FORCE_32BIT           = 0x7FFFFFFF
} WFCDeviceFilter;

typedef enum {
    /* Read-only */
    WFC_DEVICE_CLASS                        = 0x7030,
    WFC_DEVICE_ID                           = 0x7031,
    WFC_DEVICE_FORCE_32BIT                  = 0x7FFFFFFF
} WFCDeviceAttrib;

typedef enum {
    WFC_DEVICE_CLASS_FULLY_CAPABLE          = 0x7040,
    WFC_DEVICE_CLASS_OFF_SCREEN_ONLY        = 0x7041,
    WFC_DEVICE_CLASS_FORCE_32BIT            = 0x7FFFFFFF
} WFCDeviceClass;

typedef enum {
    /* Read-only */
    WFC_CONTEXT_TYPE                        = 0x7051,
    WFC_CONTEXT_TARGET_HEIGHT               = 0x7052,
    WFC_CONTEXT_TARGET_WIDTH                = 0x7053,
    WFC_CONTEXT_LOWEST_ELEMENT              = 0x7054,

    /* Read-write */
    WFC_CONTEXT_ROTATION                    = 0x7061,
    WFC_CONTEXT_BG_COLOR                    = 0x7062,
    WFC_CONTEXT_FORCE_32BIT                 = 0x7FFFFFFF
} WFCContextAttrib;

typedef enum {
    WFC_CONTEXT_TYPE_ON_SCREEN              = 0x7071,
    WFC_CONTEXT_TYPE_OFF_SCREEN             = 0x7072,
    WFC_CONTEXT_TYPE_FORCE_32BIT            = 0x7FFFFFFF
} WFCContextType;

typedef enum {
    /* Clockwise rotation */
    WFC_ROTATION_0                          = 0x7081,  /* default */
    WFC_ROTATION_90                         = 0x7082,
    WFC_ROTATION_180                        = 0x7083,
    WFC_ROTATION_270                        = 0x7084,
    WFC_ROTATION_FORCE_32BIT                = 0x7FFFFFFF
} WFCRotation;

typedef enum {
    WFC_ELEMENT_DESTINATION_RECTANGLE       = 0x7101,
    WFC_ELEMENT_SOURCE                      = 0x7102,
    WFC_ELEMENT_SOURCE_RECTANGLE            = 0x7103,
    WFC_ELEMENT_SOURCE_FLIP                 = 0x7104,
    WFC_ELEMENT_SOURCE_ROTATION             = 0x7105,
    WFC_ELEMENT_SOURCE_SCALE_FILTER         = 0x7106,
    WFC_ELEMENT_TRANSPARENCY_TYPES          = 0x7107,
    WFC_ELEMENT_GLOBAL_ALPHA                = 0x7108,
```

```
    WFC_ELEMENT_MASK                        = 0x7109,
    WFC_ELEMENT_FORCE_32BIT                 = 0x7FFFFFFF
} WFCElementAttrib;

typedef enum {
    WFC_SCALE_FILTER_NONE                   = 0x7151,  /* default */
    WFC_SCALE_FILTER_FASTER                 = 0x7152,
    WFC_SCALE_FILTER_BETTER                 = 0x7153,
    WFC_SCALE_FILTER_FORCE_32BIT            = 0x7FFFFFFF
} WFCScaleFilter;

typedef enum {
    WFC_TRANSPARENCY_NONE                   = 0,       /* default */
    WFC_TRANSPARENCY_ELEMENT_GLOBAL_ALPHA   = (1 << 0),
    WFC_TRANSPARENCY_SOURCE                 = (1 << 1),
    WFC_TRANSPARENCY_MASK                   = (1 << 2),
    WFC_TRANSPARENCY_FORCE_32BIT            = 0x7FFFFFFF
} WFCTransparencyType;

typedef enum {
    WFC_VENDOR                              = 0x7200,
    WFC_RENDERER                            = 0x7201,
    WFC_VERSION                             = 0x7202,
    WFC_EXTENSIONS                          = 0x7203,
    WFC_STRINGID_FORCE_32BIT                = 0x7FFFFFFF
} WFCStringID;


/* Function Prototypes */

/* Device */
WFC_API_CALL WFCint WFC_APIENTRY
    wfcEnumerateDevices(WFCint *deviceIds, WFCint deviceIdsCount,
        const WFCint *filterList) WFC_APIEXIT;
WFC_API_CALL WFCDevice WFC_APIENTRY
    wfcCreateDevice(WFCint deviceId, const WFCint *attribList)
WFC_APIEXIT;
WFC_API_CALL WFCErrorCode WFC_APIENTRY
    wfcGetError(WFCDevice dev) WFC_APIEXIT;
WFC_API_CALL WFCint WFC_APIENTRY
    wfcGetDeviceAttribi(WFCDevice dev, WFCDeviceAttrib attrib)
WFC_APIEXIT;
WFC_API_CALL WFCErrorCode WFC_APIENTRY
    wfcDestroyDevice(WFCDevice dev) WFC_APIEXIT;

/* Context */
WFC_API_CALL WFCContext WFC_APIENTRY
    wfcCreateOnScreenContext(WFCDevice dev,
        WFCint screenNumber,
        const WFCint *attribList) WFC_APIEXIT;
WFC_API_CALL WFCContext WFC_APIENTRY
    wfcCreateOffScreenContext(WFCDevice dev,
        WFCNativeStreamType stream,
        const WFCint *attribList) WFC_APIEXIT;
WFC_API_CALL void WFC_APIENTRY
    wfcCommit(WFCDevice dev, WFCContext ctx, WFCboolean wait)
WFC_APIEXIT;
```

```
WFC_API_CALL WFCint WFC_APIENTRY
    wfcGetContextAttribi(WFCDevice dev, WFCContext ctx,
        WFCContextAttrib attrib) WFC_APIEXIT;
WFC_API_CALL void WFC_APIENTRY
    wfcGetContextAttribfv(WFCDevice dev, WFCContext ctx,
        WFCContextAttrib attrib, WFCint count, WFCfloat *values)
WFC_APIEXIT;
WFC_API_CALL void WFC_APIENTRY
    wfcSetContextAttribi(WFCDevice dev, WFCContext ctx,
        WFCContextAttrib attrib, WFCint value) WFC_APIEXIT;
WFC_API_CALL void WFC_APIENTRY
    wfcSetContextAttribfv(WFCDevice dev, WFCContext ctx,
        WFCContextAttrib attrib,
        WFCint count, const WFCfloat *values) WFC_APIEXIT;
WFC_API_CALL void WFC_APIENTRY
    wfcDestroyContext(WFCDevice dev, WFCContext ctx) WFC_APIEXIT;

/* Source */
WFC_API_CALL WFCSource WFC_APIENTRY
    wfcCreateSourceFromStream(WFCDevice dev, WFCContext ctx,
        WFCNativeStreamType stream,
        const WFCint *attribList) WFC_APIEXIT;
WFC_API_CALL void WFC_APIENTRY
    wfcDestroySource(WFCDevice dev, WFCSource src) WFC_APIEXIT;

/* Mask */
WFC_API_CALL WFCMask WFC_APIENTRY
    wfcCreateMaskFromStream(WFCDevice dev, WFCContext ctx,
        WFCNativeStreamType stream,
        const WFCint *attribList) WFC_APIEXIT;
WFC_API_CALL void WFC_APIENTRY
    wfcDestroyMask(WFCDevice dev, WFCMask mask) WFC_APIEXIT;

/* Element */
WFC_API_CALL WFCElement WFC_APIENTRY
    wfcCreateElement(WFCDevice dev, WFCContext ctx,
        const WFCint *attribList) WFC_APIEXIT;
WFC_API_CALL WFCint WFC_APIENTRY
    wfcGetElementAttribi(WFCDevice dev, WFCElement element,
        WFCElementAttrib attrib) WFC_APIEXIT;
WFC_API_CALL WFCfloat WFC_APIENTRY
    wfcGetElementAttribf(WFCDevice dev, WFCElement element,
        WFCElementAttrib attrib) WFC_APIEXIT;
WFC_API_CALL void WFC_APIENTRY
    wfcGetElementAttribiv(WFCDevice dev, WFCElement element,
        WFCElementAttrib attrib, WFCint count, WFCint *values)
WFC_APIEXIT;
WFC_API_CALL void WFC_APIENTRY
    wfcGetElementAttribfv(WFCDevice dev, WFCElement element,
        WFCElementAttrib attrib, WFCint count, WFCfloat *values)
WFC_APIEXIT;
WFC_API_CALL void WFC_APIENTRY
    wfcSetElementAttribi(WFCDevice dev, WFCElement element,
        WFCElementAttrib attrib, WFCint value) WFC_APIEXIT;
WFC_API_CALL void WFC_APIENTRY
    wfcSetElementAttribf(WFCDevice dev, WFCElement element,
        WFCElementAttrib attrib, WFCfloat value) WFC_APIEXIT;
```

```
WFC_API_CALL void WFC_APIENTRY
    wfcSetElementAttribiv(WFCDevice dev, WFCElement element,
        WFCElementAttrib attrib,
        WFCint count, const WFCint *values) WFC_APIEXIT;
WFC_API_CALL void WFC_APIENTRY
    wfcSetElementAttribfv(WFCDevice dev, WFCElement element,
        WFCElementAttrib attrib,
        WFCint count, const WFCfloat *values) WFC_APIEXIT;
WFC_API_CALL void WFC_APIENTRY
    wfcInsertElement(WFCDevice dev, WFCElement element,
        WFCElement subordinate) WFC_APIEXIT;
WFC_API_CALL void WFC_APIENTRY
    wfcRemoveElement(WFCDevice dev, WFCElement element) WFC_APIEXIT;
WFC_API_CALL WFCElement WFC_APIENTRY
    wfcGetElementAbove(WFCDevice dev, WFCElement element) WFC_APIEXIT;
WFC_API_CALL WFCElement WFC_APIENTRY
    wfcGetElementBelow(WFCDevice dev, WFCElement element) WFC_APIEXIT;
WFC_API_CALL void WFC_APIENTRY
    wfcDestroyElement(WFCDevice dev, WFCElement element) WFC_APIEXIT;

/* Rendering */
WFC_API_CALL void WFC_APIENTRY
    wfcActivate(WFCDevice dev, WFCContext ctx) WFC_APIEXIT;
WFC_API_CALL void WFC_APIENTRY
    wfcDeactivate(WFCDevice dev, WFCContext ctx) WFC_APIEXIT;
WFC_API_CALL void WFC_APIENTRY
    wfcCompose(WFCDevice dev, WFCContext ctx, WFCboolean wait)
WFC_APIEXIT;
WFC_API_CALL void WFC_APIENTRY
    wfcFence(WFCDevice dev, WFCContext ctx, WFCEGLDisplay dpy,
        WFCEGLSync sync) WFC_APIEXIT;

/* Renderer and extension information */
WFC_API_CALL WFCint WFC_APIENTRY
    wfcGetStrings(WFCDevice dev,
        WFCStringID name,
        const char **strings,
        WFCint stringsCount) WFC_APIEXIT;
WFC_API_CALL WFCboolean WFC_APIENTRY
    wfcIsExtensionSupported(WFCDevice dev, const char *string)
WFC_APIEXIT;

#ifdef __cplusplus
}
#endif

#endif /* _WFC_H_ */
```

The following is an example of a platform-specific header file containing definitions that may be platform-specific.

wfcPlatform.h

```
/*****************************************************************************
 *                                                                           *
 * Sample implementation of wfcplatform.h, version 1.0, draft 22             *
 *                                                                           *
 * Copyright © 2007-2009 The Khronos Group                                   *
 *                                                                           *
 *****************************************************************************/

#ifndef _WFCPLATFORM_H_
#define _WFCPLATFORM_H_

#include <KHR/khrplatform.h>
#include <EGL/egl.h>

#ifdef __cplusplus
extern "C" {
#endif

#ifndef WFC_API_CALL
#define WFC_API_CALL KHRONOS_APICALL
#endif
#ifndef WFC_APIENTRY
#define WFC_APIENTRY KHRONOS_APIENTRY
#endif
#ifndef WFC_APIEXIT
#define WFC_APIEXIT KHRONOS_APIATTRIBUTES
#endif

#ifndef WFC_DEFAULT_SCREEN_NUMBER
#define WFC_DEFAULT_SCREEN_NUMBER (0)
#endif

typedef enum {
    WFC_FALSE               = KHRONOS_FALSE,
    WFC_TRUE                = KHRONOS_TRUE,
    WFC_BOOLEAN_FORCE_32BIT = 0x7FFFFFFF
} WFCboolean;

typedef khronos_int32_t   WFCint;
typedef khronos_float_t   WFCfloat;
typedef khronos_uint32_t  WFCbitfield;
typedef khronos_uint32_t  WFCHandle;

typedef EGLDisplay WFCEGLDisplay;
typedef void*      WFCEGLSync;   /* An opaque handle to an EGLSyncKHR
*/
typedef void*      WFCNativeStreamType;

#ifdef __cplusplus
}
#endif

#endif /* _WFCPLATFORM_H_ */
```

12 Appendix B: Filtering Behavior

The following Section is provided as guidance text to further explain the filtering behavior defined in the specification.

If two Elements, using the same source and scale factor, create touching source rectangles and these are mapped to two touching destination rectangles with the same relative position and size, the result should appear continuous across the join. ***Figure 2*** shows an example of continuous filtering where two Elements scale touching source rectangles. Note that the destination rectangle on the right has a light blue pixel despite the corresponding source rectangle not containing any blue pixels.

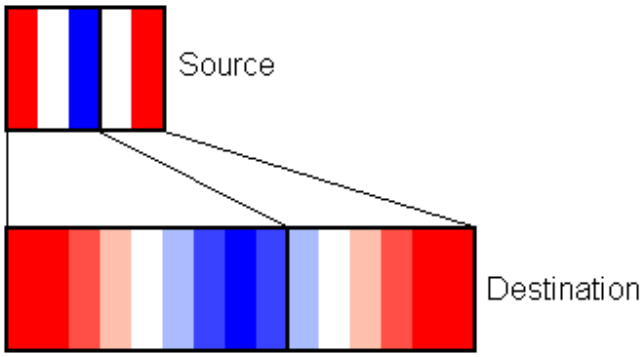


Figure 2

13 Bibliography

EGL08 Khronos Group: *EGL_KHR_reusable_sync*, Version 19, July 2009
http://www.khronos.org/registry/egl/extensions/KHR/EGL_KHR_reusable_sync.txt

KHR09 Khronos Group: khrplatform.h, Revision 7820, April 2009
http://www.khronos.org/registry/egl/api/khrplatform.h

OES07 Khronos Group: *OES_EGL_Image*, Version 4, April 2007
http://www.khronos.org/registry/gles/extensions/OES/OES_EGL_image.txt

14 Acknowledgments

This specification and the accompanying conformance test suite were developed by the Khronos OpenWF working group:

- Peter Wilson (ARM), WG Chair (from 01/2009)
- Pasi Keränen (Nokia), WG Chair (to 01/2009)
- Robert Palmer (Symbian), WFC Editor
- Steven Fischer (Motorola), WFD Editor
- Lars Remes (Symbio), CT/SI Editor
- Mikko Strandborg (Acrodea)
- Samuli Lehti (Acrodea)
- Keh-Li Sheng (Aplix)
- Ed Plowman (ARM)
- Jan-Harald Fredriksen (ARM)
- Jeremy Johnson (ARM)
- Remi Pedersen (ARM)
- Bill Licea-Kane (AMD)
- Andrej Mamona (AMD)
- Juuso Heikkila (AMD)
- Tom Longo (AMD)
- Roger Nixon (Broadcom)
- Frederic Gabin (Ericsson)
- Marcus Lorentzon (Ericsson)
- Brian Murray (Freescale)
- Mark Callow (Hi Corporation)
- Guillaume Portier (Hi Corporation)
- Micheal Green (Imagination)
- Frank Bouwer (Imagination)
- Georg Kolling (Imagination)
- Szabolcs Tolnai (Imagination)
- James Walker (NDS)
- Jani Väisänen (Nokia)
- Mika Pesonen (Nokia)
- Ari-Matti Leppanen (Nokia)
- Neil Trevett (NVIDIA)
- Kalle Raita (NVIDIA)
- Tom McReynolds (NVIDIA)
- Andre Lepine (NXP)
- Jonathan Grant (Renesas)
- Peter Brown (Renesas)
- John Leech (Self)
- Jerry Evans (Sun Microsystems)
- Jarkko Kemppainen (Symbio)
- Timo Laru (Symbio)
- Olli Vertanen (Symbio)
- Harri Kyllönen (Symbio)
- Rick Tillery (Texas Instruments)
- Tom Olsen (Texas Instruments)

15 Function Index