OpenWF Display Specification
Version 1.0
3 November 2009

Editor: Steven Fischer

Contents

1 Overview ... 1
1.1 What is Composition? ... 1
1.2 What is a Compositing Window System? ... 1
1.3 Target Users ... 2
1.4 Target Hardware ... 2
1.5 Design Philosophy ... 2
2 Definitions, Data Types, Etc. ... 4
2.1 Versioning ... 4
2.2 Definitions ... 4
2.3 Primitive Data Types ... 4
2.4 Floating Point and Integer Representations ... 6
2.5 Enumerated Data Types ... 6
2.6 Handle-based Data Types ... 7
2.7 EGL Handles ... 7
2.8 Streams ... 8
2.8.1 Functional Requirements ... 8
2.9 Attribute Lists ... 9
2.10 Coordinate Systems ... 10
2.11 Errors ... 10
2.12 Setting and Querying Attributes ... 11
2.13 Contexts & Threading ... 12
3 Devices ... 13
3.1 Enumerating Devices ... 14
3.1.1 Enumeration Filtering Attributes ... 14
3.2 Creating a Device ... 15
3.3 Destroying a Device ... 16
3.4 Committing Modifications ... 16
3.5 Device Configuration ... 18
3.5.1 Device Attributes ... 18
3.5.2 Querying Device Attributes ... 19
3.5.3 Setting Device Attributes ... 19
3.6 Asynchronous Event Notification ... 20
3.6.1 Event Containers ... 20
3.6.2 Asynchronous Notification ... 24
3.6.3 Blocking Wait Notification ... 25
3.6.4 Event Filtering ... 26
3.6.5 Event Types ... 27
4 Ports ... 31
4.1 Enumerating Ports ... 31
4.2 Creating a Port ... 32
4.3 Destroying a Port ... 33

4.4 Port Modes 34
4.4.1 Port Mode Attributes 34
4.4.2 Querying Port Modes 36
4.4.3 Querying Port Mode Attributes 37
4.4.4 Setting a Port Mode 38
4.4.5 Getting the currently set Port Mode 38
4.5 Port Configuration 39
4.5.1 Port Configuration Attributes 40
4.5.2 Querying Port Configuration Attributes 49
4.5.3 Setting Port Configuration Attributes 51
4.6 Binding a Pipeline to a Port 52
4.7 Standardized Display Data 53
4.7.1 Supported Display Data Formats 53
4.7.2 Retrieving Display Data 54
5 Pipelines 56
5.1 Pipeline Stages 56
5.1.1 Stage 1 – Color Space Conversion 56
5.1.2 Stage 2 – Crop 57
5.1.3 Stage 3 – Flip & Mirror 57
5.1.4 Stage 4 – Rotate 57
5.1.5 Stage 5 – Scale & Filter 57
5.1.6 Stage 6 – Offset 57
5.1.7 Stage 7 – Layer & Blend 57
5.2 Enumerating Pipelines 57
5.3 Creating a Pipeline 59
5.4 Destroying a Pipeline 60
5.5 Image Providers 61
5.5.1 Sources 61
5.5.2 Masks 63
5.6 Using a Pipeline 65
5.6.1 Mixed Model Pipeline Usage Considerations 65
5.6.2 Binding a Source to a Pipeline 66
5.6.3 Binding a Mask to a Pipeline 67
5.6.4 Image Transitions 68
5.7 Pipeline Configuration 69
5.7.1 Pipeline Configuration Attributes 70
5.7.2 Querying a Pipeline Configuration Attribute 78
5.7.3 Setting a Pipeline Configuration Attribute 79
5.8 Transparency 80
5.8.1 Blending Functions 81
5.8.2 Querying Transparency Combinations 82
5.8.3 Transparent Source Color Definition 83
5.9 Querying the Layer number for any Port 85

6 Extending OpenWF Display 86
6.1 Extension Naming Conventions 86
6.2 The Extension Registry 86
6.3 Using Extensions 86
6.3.1 Accessing Extensions Statically 87
6.3.2 Accessing Extensions Dynamically 87
6.4 Creating Extensions 89
7 Pre-Configuration 90
8 Image Swapping & Updating 91
8.1 Direct Hardware Refreshing 91
8.2 Limited Update Region 91
9 Appendix A: Header Files 92
10 Bibliography 104
11 Acknowledgments 105
12 Function Index 106

1 Overview

This document details the programming interface for OpenWF Display. Developed as an open standard by The Khronos Group, OpenWF Display serves as a low-level interface for display control hardware used in embedded and mobile devices.

The main goal of OpenWF Display is to provide a device-independent and vendor-neutral interface to allow control over the display control hardware. The OpenWF Display API is also designed to allow the use of OpenWF Composition for on-screen composition.

This document defines the C language binding to OpenWF Display. 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 Display functionality, and the term "user" to refer to any software that makes use of OpenWF Display.

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 control 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 Display API. Conventional applications are not intended to be direct users.

Window Systems
OpenWF Display must provide services to Window Systems to generically and efficiently allow display hardware initialization and use for OpenWF Composition on-screen rendering or direct Window System rendering, if desired.

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

1.4 Target Hardware

OpenWF Display is designed to be implementable on a wide range of display control hardware, but is mainly targeted at low-power display control hardware on systems where memory bandwidth may be a limiting factor.

OpenWF Display is designed to abstract implementations that make use of multiple display control hardware devices. On hardware platforms where multiple devices are provided by one vendor, the vendor may provide a single implementation that transparently manages all the available devices.

On hardware platforms where an OEM integrates multiple devices from independent vendors, multiple OpenWF Display implementations can exist that each drives a single device. In this case the user, which may be some OEM software, is responsible for the managing the available devices.

1.5 Design Philosophy

OpenWF Display is intended to provide a hardware abstraction layer for existing or future display control hardware blocks. Implementations of OpenWF Display are expected to be dependent on the underlying hardware, thus the overall functionality exposed will be limited by the limitations of the actual display control hardware.

All graphical and multimedia content on a system must be able to be displayed using OpenWF Display, including critical graphics that must not fail. 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.

2 Definitions, Data Types, Etc.

OpenWF Display type definitions and function prototypes are found in two header files, `wfd.h` and `wfdplatform.h`. The `wfd.h` header file contains the portable definitions and `wfdplatform.h` header file contains the few non-portable definitions that may changes from platform to platform. These files will be located in a WF subdirectory of a platform-specific header file location.

2.1 Versioning

The `<WF/wfd.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 `OPENWFD_VERSION_1_0` is defined. The version may be queried at runtime using the **wfdGetStrings** function (see section 6.3).

```
#define OPENWFD_VERSION_1_0  (1)
```

2.2 Definitions

- *Device* - an API representation of an entire display control hardware block
- *Display Control Hardware* - a hardware block which receives image data and generates an output image directed towards display hardware
- *Display Hardware* - a CRT or LCD panel type device used to show the final output image
- *Implementation* - the software and/or hardware that implements OpenWF Display functionality
- *Pipeline* – an API representation of the portion of a display control hardware block which takes a single input image and processes the image for combining, or layering, with other pipelines to create a combined output image
- *Port* – an API representation the portion of a display control hardware block which allows for the configuration of a single display hardware interface
- *User* – any software that makes use of OpenWF Display
- *Unadjusted Standard Time* – a measurement of time, in nanoseconds, since some arbitrary datum, which is fine-grained enough to expose whatever accuracy the platform allows and is defined never to decrease

2.3 Primitive Data Types

This API defines a number of primitive data types by means of C typedefs. The actual data types used are platform-specific.

WFDboolean

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

```
typedef enum
{ WFD_FALSE = KHRONOS_FALSE,
  WFD_TRUE  = KHRONOS_TRUE
} WFDboolean;
```

WFDuint8

`WFDuint8` defines an 8-bit unsigned integer, which may contain values between 0 and 255, inclusive. If `<KHR/khrplatform.h>` is available, refer to [KHR09], `WFDuint8` will be defined as `khronos_uint8_t`.

WFDint

`WFDint` defines a two's complement signed integer which is 32 bits or larger. If `<KHR/khrplatform.h>` is available, `WFDint` will be defined as `khronos_int32_t`.

WFDfloat

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

WFDbitfield

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

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

WFDtime

`WFDtime` defines a 64-bit unsigned integer representing intervals in nanoseconds (Unadjusted Standard Time). If `<KHR/khrplatform.h>` is available, `WFDtime` will be defined as `khronos_utime_nanoseconds_t`.

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

2.4 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:

WFD_MAX_INT
The macro `WFD_MAX_INT` defines the largest positive integer value that will be accepted by an implementation. `WFD_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 (`-WFD_MAX_INT`), or -16,777,216. Exceptions to this rule are explicitly noted.

WFD_MAX_FLOAT
The macro `WFD_MAX_FLOAT` defines the largest positive floating-point value that will be accepted by an implementation. `WFD_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 (`-WFD_MAX_FLOAT`), or -16,777,216.

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 6).

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

The enumerated types (apart from `WFDboolean`) defined by this API are:

- `WFDCommitType`
- `WFDDeviceAttrib`
- `WFDDisplayDataFormat`
- `WFDErrorCode`
- `WFDEventType`
- `WFDEventAttrib`
- `WFDPartialRefresh`
- `WFDPipelineConfigAttrib`
- `WFDPipelineUser`
- `WFDPortConfigAttrib`
- `WFDPortModeAttrib`
- `WFDPortType`
- `WFDPowerMode`
- `WFDScaleFilter`
- `WFDStringID`
- `WFDTransition`
- `WFDTransparency`
- `WFDTSColorFormat`

2.6 Handle-based Data Types

Handles make use of the `WFDHandle` data type. For reasons of binary compatibility between different implementations of this API on a given platform, a `WFDHandle` 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 `WFDHandle` subtypes defined in the API are:
- `WFDDevice` – a reference to a device (see Section 3)
- `WFDEvent` – a reference to an event container (see Section 3.6.1)
- `WFDPort` – a reference to a port (see Section 4)
- `WFDPortMode` – a reference to a set of port mode attributes (see Section 4.4)
- `WFDPipeline` – a reference to a pipeline (see Section 5)
- `WFDSource` – a reference to a source input provider (see Section 5.5)
- `WFDMask` – a reference to a mask input provider (see Section 5.5)

The symbol `WFD_INVALID_HANDLE` represents an invalid `WFDHandle` that is used as an error return value from functions that return a `WFDHandle`.

```
#define WFD_INVALID_HANDLE ((WFDHandle)0)
```

2.7 EGL Handles

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

A `WFDEGLImage` is an opaque handle to an `EGLImage` created using the EGL. Refer to [EGL07] for details on an `EGLImage`.

```
typedef void* WFDEGLImage;
```

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

```
typedef void* WFDEGLDisplay;
```

A `WFDEGLSync` is an opaque handle to a reusable sync object created using the EGL. Refer to [EGL09] for details on an `EGLSyncKHR`.

```
typedef void* WFDEGLSync;
```

2.8 Streams

OpenWF Display uses both `EGLImage`s and *streams* for pixel data input. 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 OpenGL ES renderer into OpenWF Display 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 Display, can receive frames without user interaction.

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

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

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

2.8.1 Functional Requirements

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

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

- WFDSource inputs

Content Provision
When a stream is connected as an input to a Pipeline, 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 Pipeline observes the contents of the backbuffer when it next renders frames in which the stream is visible. Pipeline 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.

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 Display.

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

Immutability
The following properties of streams used with OpenWF Display 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 WFDint *attribList` parameter. All attribute names in ***`attribList`*** are immediately followed by the corresponding value. The list is terminated with `WFD_NONE`. ***`attribList`*** may be `NULL` or empty (first attribute is `WFD_NONE`). Providing a non-`NULL` ***`attribList`*** that is not terminated with `WFD_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 `WFD_ERROR_BAD_ATTRIBUTE` error will be generated. If *`attribList`* contains an invalid attribute value for a legal attribute, a `WFD_ERROR_ILLEGAL_ARGUMENT` error will be generated.

```
#define WFD_NONE  (0)
```

2.10 Coordinate Systems

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

All port coordinates are specified to be relative to the top left of the associated display hardware. Pipeline source rectangle coordinates are specified to be relative to the top left of the image they are associated with. Pipeline destination rectangle coordinates are specified to be relative to top left of the associated display hardware.

2.11 Errors

Where possible, when an API function fails it has no side effects. An error condition within a function of this API 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 `WFD_INVALID_HANDLE` instead of the requested handle. Most OpenWF Display 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 **wfdGetError** function described below. If an invalid device handle is passed to any function, the function has no effect and no error is stored[2].

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

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

[2] 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 wfdGetError.

```
typedef enum
{ WFD_ERROR_NONE                = 0,
  WFD_ERROR_OUT_OF_MEMORY       = 0x7510,
  WFD_ERROR_ILLEGAL_ARGUMENT    = 0x7511,
  WFD_ERROR_NOT_SUPPORTED       = 0x7512,
  WFD_ERROR_BAD_ATTRIBUTE       = 0x7513,
  WFD_ERROR_IN_USE              = 0x7514,
  WFD_ERROR_BUSY                = 0x7515,
  WFD_ERROR_BAD_DEVICE          = 0x7516,
  WFD_ERROR_BAD_HANDLE          = 0x7517,
  WFD_ERROR_INCONSISTENCY       = 0x7518
} WFDErrorCode;
```

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

```
WFDErrorCode wfdGetError( WFDDevice device );
```

The **wfdGetError** function returns the oldest error code provided by an API call on *device* since the previous call to **wfdGetError** on that device (or since the creation of the device). No error is indicated by a return value of 0 (WFD_ERROR_NONE). After the call, the error code is cleared to 0. If *device* is not a valid device, WFD_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 WFDint, WFDbitfield, WFDboolean, enumeration, or other integral type configuration attribute from a WFDPort, **wfdGetPortAttribi** would be used. In order to query a WFDfloat type configuration attribute from a WFDPort, **wfdGetPortAttribf** 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 WFD_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 or a `WFD_ERROR_ILLEGAL_ARGUMENT` error will be raised.

Certain attributes are read-only. Attempting to set a read-only attribute has no effect other than generating a `WFD_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 & Threading

OpenWF Display 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 Devices

All display operations and resources are ultimately associated with a device, or `WFDDevice`. A `WFDDevice` is an abstraction that typically corresponds to a single display control hardware block. Each `WFDDevice` must support one or more ports (`WFDPort`) to display hardware such as CRTs or LCD panels. These ports are used to configure settings related to how the display hardware is to be updated, but not related to the actual rendered content. Each `WFDDevice` is also expected to support zero or more graphics pipelines (`WFDPipeline`), which are used to provide the visual content to be rendered to the display hardware.

A `WFDDevice` must first be created before any port or pipeline functionality can be accessed. Only one outstanding instance of a `WFDDevice` is allowed for any specific device[3].

The following example diagram depicts a platform with a `WFDDevice` which contains one `WFDPort` and two `WFDPipelines`.

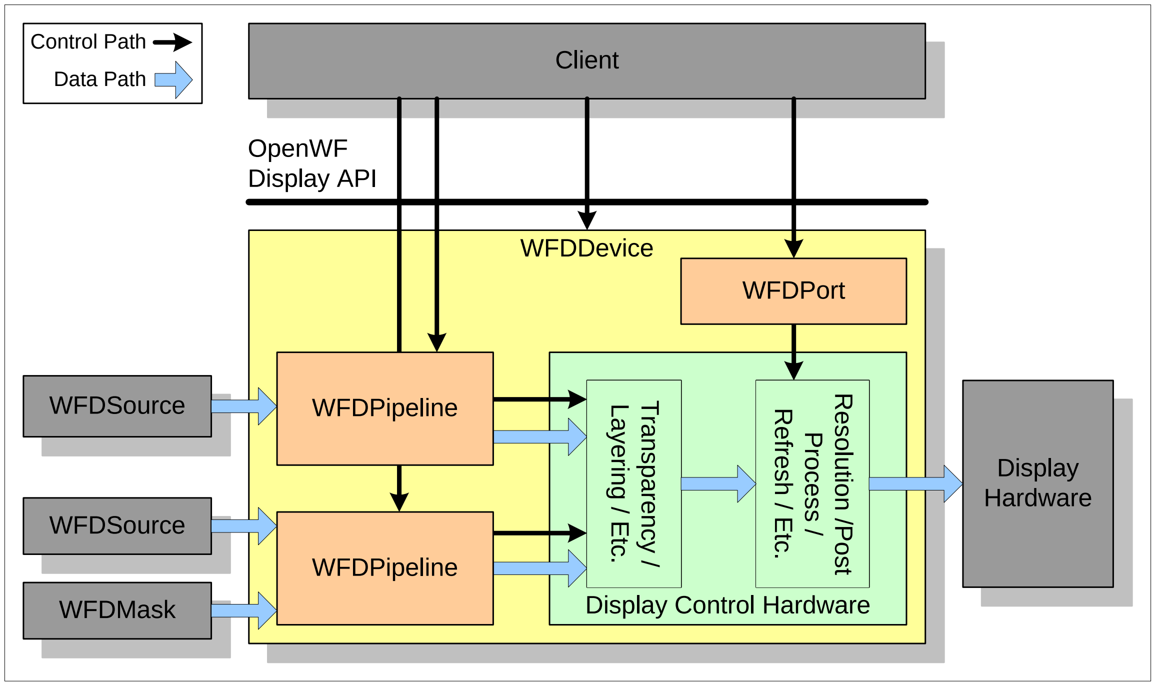

Figure 1: Example Platform

[3] It is expected that any device sharing required by the system will be managed and arbitrated by the user of this API, typically the platform Windowing System.

3.1 Enumerating Devices

The number and IDs of the available devices on the system can be retrieved by calling:

```
WFDint wfdEnumerateDevices( WFDint       *deviceIds,
                            WFDint        deviceIdsCount,
                            const WFDint *filterList );
```

The *`deviceIds`* and *`deviceIdsCount`* parameters define the caller provided array to return the device IDs into. A failure will result if *`deviceIds`* is non-`NULL` and *`deviceIdsCount`* is zero or negative.

The *`filterList`* parameter contains a list of filtering attributes which are used to control the device IDs returned by **wfdEnumerateDevices**. All attributes in *`filterList`* are immediately followed by the corresponding value. The list is terminated with `WFD_NONE`. *`filterList`* may be `NULL` or empty (first attribute is `WFD_NONE`), in which case all valid device IDs for the platform are returned. Providing a non-`NULL` *`filterList`* that is not terminated with `WFD_NONE` will result in undefined behavior. Providing an invalid filter attribute or filter attribute value will result in no device IDs being returned and a return value of zero. See section 3.1.1 for the list of available filtering attributes.

On success, the number of *`deviceIds`* elements that were populated with device ID values is returned via the function return value. Thus elements 0 through (the returned value) - 1 of *`deviceIds`* will contain valid device ID values. No more than *`deviceIdsCount`* values will be returned, even if more are available. However, if **wfdEnumerateDevices** is called with *`deviceIds`* = `NULL` then no device IDs are returned, but the total number of available device IDs is returned via the function return value.

On failure, zero is returned and *`deviceIds`* is not modified.

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

3.1.1 Enumeration Filtering Attributes

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

```
typedef enum
{ WFD_DEVICE_FILTER_PORT_ID = 0x7530
} WFDDeviceFilter;
```

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

Attribute	Type	Default
WFD_DEVICE_FILTER_PORT_ID	WFDint	WFD_INVALID_PORT_ID

Table 1: Device Filter Attributes

3.1.1.1 Port ID Filter

The `WFD_DEVICE_FILTER_PORT_ID` filter attribute is used to limit the returned device ID list to only the device which contains the port associated with the given port ID. No port ID filtering is performed if the given port ID is `WFD_INVALID_PORT_ID`.

3.2 Creating a Device

A device can be created by calling:

```
WFDDevice wfdCreateDevice( WFDint         deviceId,
                           const WFDint *attribList );
```

The *`deviceId`* parameter denotes the device to create. The *`deviceId`* value must be either a device ID retrieved using **wfdEnumerateDevices**() or `WFD_DEFAULT_DEVICE_ID`. If *`deviceId`* is `WFD_DEFAULT_DEVICE_ID`, a default device is returned. The system integrator will determine the default device.

```
#define WFD_DEFAULT_DEVICE_ID  (0)
```

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

On success, a valid `WFDDevice` handle is created and returned.

On failure, `WFD_INVALID_HANDLE` is returned.

If no device matching *`deviceId`* is available or if an out-of-memory condition exists, `WFD_INVALID_HANDLE` is returned.

Only one outstanding instance of a `WFDDevice` is allowed for any specific device. While an instance of a specific device is outstanding, created but not yet destroyed, any further attempts to create the same device will fail, resulting in `WFD_INVALID_HANDLE` being returned.

3.3 Destroying a Device

A device is destroyed by calling:

```
WFDErrorCode wfdDestroyDevice( WFDDevice device );
```

The *device* parameter denotes the device to destroy. All resources owned by *device* are deleted before this call completes. *device* is no longer a valid device handle. All port and pipeline handles created with *device* are similarly destroyed and invalidated. All resource handles associated with *device* become invalid. Any references held by *device* on external resources are removed.

All pending usages of input resources associated with *device* are completed before this call completes.

If *device* is not a valid device, WFD_ERROR_BAD_DEVICE is returned. Otherwise, WFD_ERROR_NONE is returned.

A destroyed device can be re-created given the associated device ID and a call to **wfdCreateDevice**. The handle to a re-created device should not be expected to match the previously destroyed handle.

3.4 Committing Modifications

All modification to device attributes (section 3.5) or corresponding ports (section 4) or corresponding pipelines (section 5) are not immediately applied to the associated hardware. The modifications are cached, left uncommitted, until the user explicitly applies the changes by calling[4]:

```
void wfdDeviceCommit( WFDDevice     device,
                      WFDCommitType type,
                      WFDHandle     handle );
```

The *device* parameter denotes the specific device associated with this function call.

The *type* parameter denotes the scope of the commit call. The `WFDCommitType` enumeration lists the possible commit type values.

[4] Caching and committing configuration changes in this way will cause some additional implementation overhead. The justification for this additional overhead is the potential for more seamless transitions from one scenario to another by allowing the user to make a series of changes and have them committed all at once.

```
typedef enum
{ WFD_COMMIT_ENTIRE_DEVICE         = 0x7550,
  WFD_COMMIT_ENTIRE_PORT           = 0x7551,
  WFD_COMMIT_PIPELINE              = 0x7552
} WFDCommitType;
```

The `WFD_COMMIT_ENTIRE_DEVICE` value is used to commit all of the uncommitted changes to ***`device`*** and all of the ports and pipelines associated to ***`device`***.

The `WFD_COMMIT_ENTIRE_PORT` value is used to commit all of the uncommitted changes to a specified port and all bound, or to be bound, pipelines of that port. If an attempt is made to use this commit type to commit a change that affects another port, as in bind a pipeline that is current bound to another port, then a `WFD_ERROR_INCONSISTENCY` error will be generated.

The `WFD_COMMIT_PIPELINE` value is used to commit all of the uncommitted changes for a specified pipeline.

The ***`handle`*** parameter is used to specify a port or pipeline handle for calls with a commit type of `ENTIRE_PORT` or `PIPELINE`, respectively. For calls with an `ENTIRE_DEVICE` commit type, ***`handle`*** must always be set to `WFD_INVALID_HANDLE`.

On success, this function will not return until all cached modifications, which are to be committed, are indeed committed to the hardware, with the exception of pipeline bindings with the `WFD_TRANSITION_AT_VSYNC` transition. In the case of pipeline bindings with `WFD_TRANSITION_AT_VSYNC`, the binding will complete as described in section 5.6.4.

On failure, one of the errors below will be raised.

The implementation will test for conflicts between cached modifications or between cached modifications and existing unmodified settings before any modifications are committed to the hardware. If any such conflict exists, the `WFD_ERROR_INCONSISTENCY` error is generated and the hardware is left in the pre-call state.

The implementation will also validate that the requested new configuration is supported by the hardware. If the platform hardware can not support the configuration, either due to resource (memory bandwidth, etc.) or other limitation, the `WFD_ERROR_NOT_SUPPORTED` error is generated and the hardware is left in the pre-call state.

In all cases, when **wfdDeviceCommit** returns the corresponding modification caches are cleared.

When a device is initially created, the cache of device configuration changes will be empty.

ERRORS

`WFD_ERROR_ILLEGAL_ARGUMENT`
- if *`type`* is not a valid `WFDCommitType`

`WFD_ERROR_NOT_SUPPORTED`
- if any hardware resource limitations exist

`WFD_ERROR_BUSY`
- if a call to **wfdDeviceCommit** is currently executing

`WFD_ERROR_BAD_HANDLE`
- if *`handle`* does not match the requirements of *`type`*, as specified above

`WFD_ERROR_INCONSISTENCY`
- if any configuration conflicts exist within the device or corresponding ports or corresponding pipelines

3.5 Device Configuration

For each `WFDDevice` there can be a number of attributes for status information and configuration. These attributes allow for the configuration of display control device features.

Configuration attribute changes are not immediately applied to the hardware. They are cached until **wfdDeviceCommit** is called. All retrieved attributes will reflect the cached values, if a cached value exists for the attribute.

Unless otherwise noted, the read-only attributes of the device configuration are static while associated device handle, `WFDDevice`, is valid.

3.5.1 Device Attributes

A device contains only one attribute[5]. The `WFDDeviceAttrib` enumeration lists this device attribute.

```
typedef enum
{ WFD_DEVICE_ID                         = 0x7560
} WFDDeviceAttrib;
```

[5] The device attribute list and getter/setter functions are also defined to allow for ease of adding extensions.

The storage type, default value, and read-writable state for all device attributes are listed in ***Table* 2**.

Attribute	Storage Type	R/W	Default
`WFD_DEVICE_ID`	`WFDint`	`R`	

Table 2: Device Attributes

3.5.1.1 Device ID

The `WFD_DEVICE_ID` attribute denotes the ID of the device. This value will not be `WFD_DEFAULT_DEVICE_ID`.

Accessors: **wfdGetDeviceAttribi**

3.5.2 Querying Device Attributes

A device attribute can be queried by calling:

```
WFDint wfdGetDeviceAttribi( WFDevice       device,
                            WFDDeviceAttrib attrib );
```

The *`device`* parameter is the handle to the device to retrieve the attribute from.

The *`attrib`* parameter denotes the attribute to retrieve and return via the function return value.

On success, the value of *`attrib`* for *`device`* is returned via the function return value.

On failure, one of the errors below will be raised.

ERRORS

`WFD_ERROR_BAD_ATTRIBUTE`
- if *`attrib`* is not a valid device attribute
- if *`attrib`* does not permit the use of the accessor

3.5.3 Setting Device Attributes

A writable device attribute can be modified by calling:

```
void wfdSetDeviceAttribi( WFDDevice       device,
                          WFDDeviceAttrib attrib,
                          WFDint          value );
```

The *`device`* parameter is the handle to the device to update the attribute of.

The `attrib` parameter denotes the attribute to modify to the new `value` parameter setting.

On success, the value of `attrib` for `device` is updated to `value`.

On failure, the device attrib value will not be modified and one of the errors below will be raised.

ERRORS

WFD_ERROR_ILLEGAL_ARGUMENT
- if `value` is out-of-range for `attrib`

WFD_ERROR_BAD_ATTRIBUTE
- if `attrib` is not a valid device attribute
- if `attrib` denotes a read-only attribute
- if `attrib` does not permit the use of the accessor

3.6 Asynchronous Event Notification

A number of asynchronous events are available to the user. The events are exposed per device, thus the user must interface with each device that events are desired from.

3.6.1 Event Containers

An Event Container is a wrapper for the event queues, settings, and other related data needed to provide event notifications to a single client. Each client that desires event notifications must have there own event container.

After creation, the event container provides the context, along with the device handle, for all of the remaining event related functions. The event container is used when selecting which events notification is desired for, when waiting for or retrieving an event, and when read individual attributes of an event occurrence.

3.6.1.1 Creating an Event Container

An event container can be created by calling:

```
WFDEvent wfdCreateEvent( WFDDevice     device,
                         const WFDint *attribList );
```

The `device` parameter denotes the specific device associated with this function call.

The `attribList` parameter is defined in section 2.9. See section 3.6.1.3 for the list of configuration attributes.

On success, a valid `WFDEvent` handle is created and returned.

On failure, `WFD_INVALID_HANDLE` is returned and one of the errors below will be raised.

ERRORS

`WFD_ERROR_OUT_OF_MEMORY`
- if lack of sufficient memory prevents the creation of the `WFDEvent`

`WFD_ERROR_ILLEGAL_ARGUMENT`
- if *`attribList`* contains an invalid value for a valid attribute

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

3.6.1.2 Destroying an Event Container

An event container can be destroyed by calling:

```
void wfdDestroyEvent( WFDDevice device,
                      WFDEvent  event );
```

The *`device`* parameter denotes the specific device associated with this function call.

The *`event`* parameter denotes the event container to destroy. Following this call, all resources associated with *`event`* are marked for deletion as soon as possible. *`event`* is no longer a valid event container handle.

ERRORS

`WFD_ERROR_BAD_HANDLE`
- if *`event`* is not a valid `WFDEvent` for *`device`*

3.6.1.3 Event Container Attributes

An event container surrounds a set of values related to the event container configuration and each of events that can occur on the platform. The event container attributes are updated to reflect the most recent event retrieved during the **wfdDeviceEventWait** call, see section 3.6.3 below.

The `WFDEventAttrib` enumeration lists the event container attributes.

```
typedef enum
{ /* Configuration Attributes */
  WFD_EVENT_PIPELINE_BIND_QUEUE_SIZE        = 0x75C0,

  /* Generic Event Attributes */
  WFD_EVENT_TYPE                            = 0x75C1,

  /* Port Attach Event Attributes */
  WFD_EVENT_PORT_ATTACH_PORT_ID             = 0x75C2,
  WFD_EVENT_PORT_ATTACH_STATE               = 0x75C3,

  /* Port Protection Event Attributes */
  WFD_EVENT_PORT_PROTECTION_PORT_ID         = 0x75C4,

  /* Pipeline Bind Complete Event Attributes */
  WFD_EVENT_PIPELINE_BIND_PIPELINE_ID       = 0x75C5,
  WFD_EVENT_PIPELINE_BIND_SOURCE            = 0x75C6,
  WFD_EVENT_PIPELINE_BIND_MASK              = 0x75C7,
  WFD_EVENT_PIPELINE_BIND_QUEUE_OVERFLOW    = 0x75C8
} WFDEventAttrib;
```

The storage type, default value, and read-writable state for all event container attributes are listed in ***Table 3***.

Attribute	Storage Type	R/W	Default
WFD_EVENT_PIPELINE_BIND_QUEUE_SIZE	WFDint	R(W)	See 3.6.1.3.1
WFD_EVENT_TYPE	WFDEventType	R	▨
WFD_EVENT_PORT_ATTACH_PORT_ID	WFDint	R	▨
WFD_EVENT_PORT_ATTACH_STATE	WFDboolean	R	▨
WFD_EVENT_PORT_PROTECTION_PORT_ID	WFDint	R	▨
WFD_EVENT_PIPELINE_BIND_PIPELINE_ID	WFDint	R	▨
WFD_EVENT_PIPELINE_BIND_SOURCE	WFDSource	R	▨
WFD_EVENT_PIPELINE_BIND_MASK	WFDMask	R	▨
WFD_EVENT_PIPELINE_BIND_QUEUE_OVERFLOW	WFDboolean	R	▨

Table 3: Event Container Attributes

The retrieval of event container attributes is performed by calling:

```
WFDint wfdGetEventAttribi( WFDDevice      device,
                           WFDEvent       event,
                           WFDEventAttrib attrib );
```

The `device` and `event` parameters denote the specific device and event container associated with this function call.

The `attrib` parameter denotes the desired event container attribute to retrieve.

On success, the value of *`attrib`* for *`event`* is returned via the function return value.

On failure, one of the errors below will be raised.

The **wfdGetEventAttribi** accessor is used to retrieve all event attributes.

ERRORS

`WFD_ERROR_ILLEGAL_ARGUMENT`
- if *`attrib`* is not a valid attribute for the current event type

`WFD_ERROR_BAD_ATTRIBUTE`
- if *`attrib`* is not a valid event container attribute

`WFD_ERROR_BAD_HANDLE`
- if *`event`* is not a valid `WFDEvent` for *`device`*

3.6.1.3.1 Pipeline Bind Queue Size

The `WFD_EVENT_PIPELINE_BIND_QUEUE_SIZE` attribute is used both to configure the pipeline source & mask bind event queue size when the event container is being created and to read back the size after creation.

During event container creation, this attribute can be configured via the *`attribList`* parameter of the **wfdCreateEvent** call. The value associate with the attribute will denote the depth, number of events that can be queued, of the pipeline source & mask bind event queue. The default value for this attribute is defined in section 3.6.5.4. If this attribute is set to 0 or negative, no pipeline source or mask bind events will occur.

After creation, the pipeline source & mask bind event queue size is a read only attribute.

3.6.1.3.2 Event Type

The `WFD_EVENT_TYPE` attribute denotes the event type of the most recently retrieved event or `WFD_EVENT_NONE` if the last retrieved event was a non-event. At creation time of an event container, the `WFD_EVENT_TYPE` attribute will be set to `WFD_EVENT_NONE.`

3.6.1.3.3 Other Event Attributes

The remaining event container attributes can only be read when the `WFD_EVENT_TYPE` attribute denotes the event type that they are related to. See section 3.6.5 for details of the event types that the attributes are related to and the valid values an attribute can hold. A `WFD_ERROR_ILLEGAL_ARGUMENT` error will be raised if an attempt is made to retrieve an attribute that does not relate to the current event type.

3.6.2 Asynchronous Notification

Asynchronous notification of device events is accomplished via a `WFDEGLSync` object, typically a reusable EGL Sync Object. The user must first create the `WFDEGLSync` object and pass the object to the device from which the asynchronous notification is desired. The same `WFDEGLSync` object may be passed to multiple devices, if desired. The `WFDEGLSync` object is passed to the device by calling:

```
void wfdDeviceEventAsync( WFDDevice     device,
                          WFDEvent      event,
                          WFDEGLDisplay display,
                          WFDEGLSync    sync);
```

The *`device`* and *`event`* parameters denote the specific device and event container, respectively, associated with this function call.

The *`sync`* parameter is a `WFDEGLSync` handle which is to be used to report events queued to *`event`* from *`device`*. The *`display`* parameter provides context for *`sync`* when interacting with EGL. The `EGL_SYNC_TYPE_KHR` attribute of *`sync`* must be equal to `EGL_SYNC_REUSABLE_KHR`. *`sync`* must have been created for *`display`*. If *`display`* does not match the `EGLDisplay` used to create *`sync`*, the behavior is undefined.

On success, *`sync`* will be stored and used to report the next event queued to *`event`* from *`device`*. Any previously stored `WFDEGLSync` object will be overwritten. If *`sync`* is equal to `WFD_INVALID_SYNC` then no new `WFDEGLSync` object is stored, but any existing `WFDEGLSync` object is still cleared.

```
#define WFD_INVALID_SYNC ((WFDEGLSync)0)
```

On failure, the previously stored `WFDEGLSync` object is not changed and one of the errors below will be raised.

After the `WFDEGLSync` handle is stored, it will used to signal the user when the next event is queued to *`event`*, or immediately if *`event`* currently has an event queued. A stored `WFDEGLSync` handle will only be signaled once. After each signal occurs the client must pass in the `WFDEGLSync` handle again. The user is responsible for un-signalling the sync object, as necessary, after each event.

Asynchronous notification is most useful when the user wishes to handle events from multiple devices within the same thread context. In this case, the user would pass the same `WFDEGLSync` object to each of the multiple devices and pend on the signal of the `WFDEGLSync` object. When signaled, the user would poll each device for one or more events, via **wfdDeviceEventWait**, thus handling any queued events. After handling all queued events, the user would pass the same `WFDEGLSync` handle to the devices and pend on further signals.

When this API is called and an existing `WFDEGLSync` object is replaced, this existing `WFDEGLSync` object is not signalled or changed in any way when it is replaced. It is left up to the user to manage the `WFDEGLSync` objects it provides and properly release any threads pending on replaced `WFDEGLSync` objects.

ERRORS

`WFD_ERROR_ILLEGAL_ARGUMENT`
- if *`display`* contains an invalid `WFDEGLDisplay` object
- if *`sync`* contains an invalid `WFDEGLSync` object
- if *`sync`*'s `EGL_SYNC_TYPE_KHR` is not `EGL_SYNC_REUSABLE_KHR`

`WFD_ERROR_BAD_HANDLE`
- if *`event`* is not a valid `WFDEvent` for *`device`*

3.6.3 Blocking Wait Notification

Blocking the calling thread waiting for an asynchronous device event is done by calling:

```
WFDEventType wfdDeviceEventWait( WFDDevice device,
                                 WFDEvent  event,
                                 WFDtime   timeout );
```

The *`device`* and *`event`* parameter denote the specific device and event container, respectively, associated with this function call.

The *`timeout`* parameter defines the length of time, in nanoseconds, to wait for an event to occur. If *`timeout`* is zero, **wfdDeviceEventWait** will not block, instead it will return immediately with the current event status. If *`timeout`* is `WFD_FOREVER`, **wfdDeviceEventWait** will not timeout.

```
#define WFD_FOREVER (0xFFFFFFFFFFFFFFFF)
```

The event container will be used to return the event specific information in. The event container handle must be created in advance via **wfdCreateEvent**.

On success, the return value will denote the event that occurred and *`event`* will contain the attributes specific to the event that occurred. The `WFD_EVENT_TYPE` attribute will be set to the same value as the return value. The remaining attributes are event specific and will contain data related to the event that occurred. Refer to the individual event descriptions below for exact usage.

On failure, the return value will be `WFD_EVENT_INVALID`, the *`event`* is not modified, and one of the errors below will be raised.

The possible asynchronous events are listed in the `WFDEventType` enumeration.

```
typedef enum
{ WFD_EVENT_INVALID                          = 0x7580,
  WFD_EVENT_NONE                             = 0x7581,
  WFD_EVENT_DESTROYED                        = 0x7582,
  WFD_EVENT_PORT_ATTACH_DETACH               = 0x7583,
  WFD_EVENT_PORT_PROTECTION_FAILURE          = 0x7584,
  WFD_EVENT_PIPELINE_BIND_SOURCE_COMPLETE = 0x7585,
  WFD_EVENT_PIPELINE_BIND_MASK_COMPLETE   = 0x7586
} WFDEventType;
```

Events will be queued for later retrieval by the user. The specific queuing requirements for each individual event is defined in the below event descriptions.

Only one waiting thread per event container is allowed. Calls to **wfdDeviceEventWait** when a thread is already waiting will result in a `WFD_ERROR_NOT_SUPPORTED` error.

When a non-filtered event occurs, the waiting thread will receive the event.

ERRORS

`WFD_ERROR_NOT_SUPPORTED`
- if *`event`* already has a waiting thread

`WFD_ERROR_BAD_HANDLE`
- if *`event`* is not a valid `WFDEvent` for *`device`*

3.6.4 Event Filtering

Events can be filtered such that only certain events will be queued to an event container and reported to the user. Event filtering is accomplished by providing the event container with a list of the desired events via calling:

```
void wfdDeviceEventFilter( WFDDevice           device,
                           WFDEvent            event,
                           const WFDEventType *filter );
```

The `device` and `event` parameter denote the specific device and event container, respectively, associated with this function call.

The `filter` parameter contains a list of the `WFDEventType` values which denote the events to be queued to `event` and reported to the user. This list must be terminated with `WFD_NONE`. By default, all events are queued and reported to the user. If `filter` is `NULL`, no filtering will be performed, thus the default behavior will occur. The `WFD_EVENT_NONE` and `WFD_EVENT_DESTROYED` events can not be disabled by filtering.

On success, only the events denoted by `filter` will be queued to `event` and reported to the user.

On failure, the previous event filter for `event` is not changed and one of the errors below will be raised.

ERRORS

`WFD_ERROR_ILLEGAL_ARGUMENT`
- if `filter` contains an invalid `WFDEventType` value

`WFD_ERROR_BAD_HANDLE`
- if `event` is not a valid `WFDEvent` for `device`

3.6.5 Event Types

A description of the each the event type is contained in the sections below.

3.6.5.1 No Event

The `WFD_EVENT_NONE` event occurs for one of the following reasons:

- A call to **wfdDeviceEventWait** timed out waiting for an event to occur.
- A call to **wfdDeviceEventWait** with a zero timeout value was made and no event was in the event queue.
- By default if **wfdGetEventAttribi** is called before **wfdDeviceEventWait**.

There are no event container attributes that are associated with this event. There is no requirement for an event queue for this event.

3.6.5.2 Destroyed Event

The destroyed event occurs when the event container or parent device is destroyed while the client is waiting on an event. For a user waiting via the blocking wait, the `WFD_EVENT_DESTROYED` event is returned by **wfdDeviceEventWait**. For a user waiting asynchronously, the `WFDEGLSync` object will be signalled. In both cases, at the time of the event the `WFDEvent` handle is no longer valid, so any attempt to use the handle will result in a `WFD_ERROR_BAD_HANDLE` error. If a destroyed device caused the event, the related device handle is also invalid at the time of the event.

3.6.5.3 Port Attach/Detach Event

The port attach/detach event occurs when a device is attached to or detached from a detachable port. See section 4.5.1.3 for details.

When this event occurs the following attributes of the event container are updated and available to be retrieved:

- The `WFD_EVENT_PORT_ATTACH_PORT_ID` attribute will be set to the port ID of the port this event is related to.
- The `WFD_EVENT_PORT_ATTACH_STATE` attribute will reflex the new attachment state of the port. Specifically, the attribute will be set to `WFD_TRUE` if the port is newly attached or `WFD_FALSE` if the port is newly detached.

The event queue requirement for the port attach/detach event are:

- A queue with a depth equal to the number of detachable ports available on the device.
- Only the most recent event from each detachable port will exist on the queue.

3.6.5.4 Port Protection Failure Event

The port protection failure event occurs when a secure port is determined to no longer be secure. See section 4.5.1.15 for details.

When this event occurs the following attribute of the event container is updated and available to be retrieved:

- The `WFD_EVENT_PORT_PROTECTION_PORT_ID` attribute will be set to the port ID of the port this event is related to.

The event queue requirements for the protection failure event are:

- A queue with a depth equal to the number of ports available on the device.
- Only the most recent event from each port will exist on the queue.

3.6.5.5 Pipeline Bind Source Complete Event

The pipeline bind source complete event occurs when a cached pipeline bind source transition completes. See section 5.6.4 for details.

When this event occurs the following attributes of the event container are updated and available to be retrieved:

- The `WFD_EVENT_PIPELINE_BIND_PIPELINE_ID` attribute will be set to the pipeline ID of the pipeline this event is related to.
- The `WFD_EVENT_PIPELINE_BIND_SOURCE` attribute will be set to the previously bound source handle, `WFDSource`. If no previous binding existed, this attribute will be set to `WFD_INVALID_HANDLE`.
- The `WFD_EVENT_PIPELINE_BIND_QUEUE_OVERFLOW` attribute will be set to `WFD_TRUE` if the bind complete event queue overflowed near this event. See below.

The combined event queue requirements for the pipeline bind source & mask complete events are:

- The depth of the queue is client configurable, see section 3.6.1.3.1. The default queue size is the sum of the largest refresh rates from each port on the device times the number of pipelines on the device. For example, if a device contains two ports and three pipelines, and both ports maximum refresh rate is 30 Hz, then the default queue depth will be 180, which is ((30 + 30) * 3).
- If the event queue is full when a new entry is to be queued the `WFD_EVENT_ WFD_EVENT_PIPELINE_BIND_QUEUE_OVERFLOW` attribute of the last entry on the queue will be set to `WFD_TRUE`[6].

3.6.5.6 Pipeline Bind Mask Complete Event

The pipeline bind mask complete event occurs when a pipeline bind mask transition completes. See section 5.6.4 for details.

When this event occurs the following attributes of the event container are updated and available to be retrieved:

- The `WFD_EVENT_PIPELINE_BIND_PIPELINE_ID` attribute will be set to the pipeline ID of the pipeline this event is related to.

[6] The overflow attribute is intended to provide the user with an indication that an overflow condition has occurred. It is left up to the user to determine how to respond to the condition.

- The `WFD_EVENT_PIPELINE_BIND_MASK` attribute will be set to the previously bound mask handle, `WFDMask`. If no previous binding existed, this attribute will be set to `WFD_INVALID_HANDLE`.
- The `WFD_EVENT_PIPELINE_BIND_QUEUE_OVERFLOW` attribute will be set to `WFD_TRUE` if the bind complete event queue overflowed near this event. See the queue requirements for details.

The event queue requirements for the pipeline bind mask complete event are combined with the pipeline bind source complete event. See the requirements in the above section.

4 Ports

The output of a display control device (`WFDDevice`) is content rendered to display hardware such as an LCD panel, a CRT, or similar hardware. A `WFDPort`, which is associated with a specific `WFDDevice`, is an abstraction of the display control device output to which display hardware may be attached. A `WFDPort` provides a means for configuring settings related to how the display hardware is to be updated, but not related to the actual rendered content. These settings include items such as: resolution, refresh rate, gamma, and others.

A single `WFDDevice` must support at least one `WFDPort`.

4.1 Enumerating Ports

The number and IDs of the available ports of a device can be retrieved by calling:

```
WFDint wfdEnumeratePorts( WFDDevice     device,
                          WFDint       *portIds,
                          WFDint        portIdsCount
                          const WFDint *filterList );
```

The *`device`* parameter denotes which display control device to retrieve the IDs from.

The *`portIds`* and *`portIdsCount`* parameters define the caller provided array to return the IDs into.

The *`filterList`* parameter contains a list of filtering attributes which are used to control the port IDs returned by **wfdEnumeratePorts**. All attributes in *`filterList`* are immediately followed by the corresponding value. The list is terminated with `WFD_NONE`. *`filterList`* may be `NULL` or empty (first attribute is `WFD_NONE`), in which case all valid port IDs for the platform are returned. Providing a non-`NULL` *`filterList`* that is not terminated with `WFD_NONE` will result in undefined behavior. Providing an invalid filter attribute or filter attribute value will result in no port IDs being returned and a return value of zero. No valid filtering attributes are defined for *`filterList`* in this specification.

On success, the number of *`portIds`* elements that were populated with port ID values is returned via the function return value. Thus elements 0 through (the returned value) - 1 of *`portIds`* will contain valid ID values. No more than *`portIdsCount`* values will be returned, even if more are available. However, if **wfdEnumeratePorts** is called with *`portIds`* `= NULL` then no IDs are returned, but the total number of available IDs is returned via the function return value.

On failure, zero is returned and *`portIds`* is not modified. Additionally, one of the errors below will be raised.

The port IDs should not be expected to be contiguous. A port ID equal to `WFD_INVALID_PORT_ID` will not be returned.

```
#define WFD_INVALID_PORT_ID (0)
```

The list of available ports for a device is static.

A port with detached display hardware will still be listed and creatable. See section 4.5.1.3 for details on detachable ports.

ERRORS

`WFD_ERROR_ILLEGAL_ARGUMENT`
- if *`portIdsCount`* is invalid, zero or negative

4.2 Creating a Port

A port can be created by calling:

```
WFDPort wfdCreatePort( WFDDevice     device,
                       WFDint        portId,
                       const WFDint *attribList );
```

The *`device`* parameter is the handle to the device that *`portId`* is associated with.

The *`portId`* parameter denotes the port to create. This value should be a port ID retrieved using **wfdEnumeratePorts**().

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

On success, a `WFDPort` representing *`portId`* is created and a valid handle is returned.

On failure, `WFD_INVALID_HANDLE` is returned and one of the errors below will be raised.

Only one outstanding instance of a `WFDPort` is allowed for any specific port. While an instance of a specific port is outstanding, created but not yet destroyed, any further attempts to create the same port will fail, resulting with a `WFD_ERROR_IN_USE` error and `WFD_INVALID_HANDLE` being returned.

ERRORS

`WFD_ERROR_ILLEGAL_ARGUMENT`
- if *`portId`* is not a valid port ID for *`device`*

`WFD_ERROR_BAD_ATTRIBUTE`
- if *`attribList`* contains invalid attributes or invalid attribute values

`WFD_ERROR_OUT_OF_MEMORY`
- if lack of sufficient memory prevents the creation of the `WFDPort`

`WFD_ERROR_IN_USE`
- if a `WFDPort` associated with *`portId`* has already been created

4.3 Destroying a Port

A port is destroyed by calling:

```
void wfdDestroyPort( WFDDevice device,
                     WFDPort   port );
```

The *`device`* parameter denotes the specific device associated with this function call.

The *`port`* parameter denotes the display port to destroy. Following this call, all resources associated with port are marked for deletion as soon as possible. The port handle, *`port`*, is no longer valid. All bindings to pipelines are released. All resource handles associated with *`port`* become invalid. Any references held by *`port`* on external resources are removed.

All pending usages of input resources associated with *`port`* are completed before this call completes.

A destroyed port can be re-created given the associated port ID and a call to **wfdCreatePort**. The handle to a re-created port should not be expected to match the previously destroyed handle.

ERRORS

`WFD_ERROR_BAD_HANDLE`
- if *`port`* is not a valid `WFDPort` for *`device`*

4.4 Port Modes

For each `WFDPort`, an operating mode must be selected prior to being used to display content. A port mode, `WFDPortMode`, consists of a collection of read only attributes that define the operating parameters of the port for the given mode. While display hardware is attached, each `WFDPort` will support one or more modes which can be individually queried. If no display hardware is attached, no port modes will be available.

Setting the port mode, via **wfdSetPortMode**, will not be immediately applied to the hardware. The new port mode will be cached until **wfdDeviceCommit** is called for the corresponding port. The retrieved current mode, via **wfdGetCurrentPortMode**, will reflect the cached mode, if a cached mode exists.

The number of available port modes and the contents of those port modes are static only while display hardware is attached. See section 4.5.1.3 for attach/detach details.

4.4.1 Port Mode Attributes

The `WFDPortModeAttrib` enumeration lists the port mode attributes.

```
typedef enum
{ WFD_PORT_MODE_WIDTH                   = 0x7600,
  WFD_PORT_MODE_HEIGHT                  = 0x7601,
  WFD_PORT_MODE_REFRESH_RATE            = 0x7602,
  WFD_PORT_MODE_FLIP_MIRROR_SUPPORT     = 0x7603,
  WFD_PORT_MODE_ROTATION_SUPPORT        = 0x7604,
  WFD_PORT_MODE_INTERLACED              = 0x7605
} WFDPortModeAttrib;
```

The storage type for all port mode attributes are listed in ***Table 4***.

Attribute	Storage Type
WFD_PORT_MODE_WIDTH	WFDint
WFD_PORT_MODE_HEIGHT	WFDint
WFD_PORT_MODE_REFRESH_RATE	WFDfloat
WFD_PORT_MODE_FLIP_MIRROR_SUPPORT	WFDboolean
WFD_PORT_MODE_ROTATION_SUPPORT	WFDRotationSupport
WFD_PORT_MODE_INTERLACED	WFDboolean

Table 4: Port Mode Attributes

4.4.1.1 Resolution

The `WFD_PORT_MODE_WIDTH` and `WFD_PORT_MODE_HEIGHT` attributes define the resolution of the port in the given mode. The width and height values are in pixel units.

Accessors: **wfdGetPortModeAttribi**

4.4.1.2 Refresh Rate

The `WFD_PORT_MODE_REFRESH_RATE` attribute defines the rate at which the visual device will be refreshed in terms of refreshes per second[7]. For interlaced refresh devices, this value denotes the rate of the half screen (odd or even lines) updates. For some types of display hardware, such as self-refreshing smart display panels, this rate denotes the maximum rate that the hardware can be updated with visible results.

Accessors: **wfdGetPortModeAttribi, wfdGetPortModeAttribf**

4.4.1.3 Flip & Mirror Support

The `WFD_PORT_MODE_FLIP_MIRROR_SUPPORT` attribute denotes whether flipping (inverting the displayed image top-to-bottom) and mirroring (inverting the displayed image left-to-right) are supported by the port in the given mode. If this attribute is `WFD_TRUE`, then both the `WFD_PORT_FLIP` and `WFD_PORT_MIRROR` configuration attributes can be used to enable or disable these features, see section 4.5.1.8 and 4.5.1.9 for details.

Accessors: **wfdGetPortModeAttribi**

4.4.1.4 Rotation Support

The `WFD_PORT_MODE_ROTATION_SUPPORT` attribute denotes whether rotation is supported by the port in the given mode. This attribute is in the form of an enumeration, `WFDRotationSupport`.

```
typedef enum
{ WFD_ROTATION_SUPPORT_NONE    = 0x76D0,
  WFD_ROTATION_SUPPORT_LIMITED = 0x76D1
} WFDRotationSupport;
```

The "`LIMITED`" case denotes support for clockwise rotation values of 0, 90, 180, and 270 degrees. The "`NONE`" case denotes that the only supported rotation value is 0 degrees, hence no rotation support.

Accessors: **wfdGetPortModeAttribi**

[7] Note that if OpenWF Composition is in use, this refresh rate is not expected to be used for content update rate limiting since OpenWF Composition contains an independent means of content update rate limiting.

4.4.1.5 Interlaced

The `WFD_PORT_MODE_INTERLACED` attribute denotes whether or not the port output signal is interlaced. If this attribute is set to `WFD_TRUE`, then the output signal is in an interlaced form, else it is not. For port types for which this property does not apply, the attribute will be defaulted to `WFD_FALSE`.

Accessors: **wfdGetPortModeAttribi**

4.4.2 Querying Port Modes

A list of the available `WFDPortMode` port modes can be queried by calling:

```
WFDint wfdGetPortModes( WFDDevice    device,
                        WFDPort      port,
                        WFDPortMode *modes,
                        WFDint       modesCount );
```

The *`device`* and *`port`* parameters denote the specific device and port, respectively, associated with this function call.

The *`modes`* and *`modesCount`* parameters define the caller provided array to return the modes into.

On success, the number of *`modes`* elements that were populated with valid `WFDPortMode` elements is returned via the function return value. Thus elements 0 through (the returned value) - 1 of *`modes`* will contain valid `WFDPortMode` elements. No more than *`modesCount`* elements will be returned, even if more are available. However, if **wfdGetPortModes** is called with *`modes`* = `NULL` then no modes are returned, but the total number of available modes is returned via the function return value.

On failure, zero is returned, *`modes`* is not modified, and one of the errors below will be raised:

ERRORS

`WFD_ERROR_ILLEGAL_ARGUMENT`
- if *`modesCount`* is invalid, zero or negative

`WFD_ERROR_NOT_SUPPORTED`
- if *`port`* is not attached

`WFD_ERROR_BAD_HANDLE`
- if *`port`* is not a valid `WFDPort` for *`device`*

4.4.3 Querying Port Mode Attributes

A port mode attribute can be queried by calling one of the following functions:

```
WFDint wfdGetPortModeAttribi( WFDDevice         device,
                              WFDPort           port,
                              WFDPortMode       mode,
                              WFDPortModeAttrib attrib );

WFDfloat wfdGetPortModeAttribf( WFDDevice         device,
                                WFDPort           port,
                                WFDPortMode       mode,
                                WFDPortModeAttrib attrib );
```

The *`device`* and *`port`* parameters denote the specific device and port, respectively, associated with this function call.

The *`mode`* parameter denotes the port mode to retrieve *`attrib`* from.

On success, the value of *`attrib`* for *`mode`* is returned via the function return value.

On failure, one of the errors below will be raised.

ERRORS

`WFD_ERROR_ILLEGAL_ARGUMENT`
- if *`mode`* is not a valid mode for *`port`*

`WFD_ERROR_BAD_HANDLE`
- if *`port`* is not a valid `WFDPort` for *`device`*

`WFD_ERROR_BAD_ATTRIBUTE`
- if *`attrib`* is not a valid port mode attribute
- if *`attrib`* does not permit the use of the accessor

4.4.4 Setting a Port Mode

A specific mode can be set for a port by calling:

```
void wfdSetPortMode( WFDDevice    device,
                     WFDPort      port,
                     WFDPortMode  mode );
```

The *`device`* and *`port`* parameters denote the specific device and port, respectively, associated with this function call.

The *`mode`* parameter is the specific port mode to set.

On success, the setting of *`mode`* to *`port`* will be cached. The change will be applied to the hardware when **wfdDeviceCommit** is called on *`port`*.

On failure, the mode of *`port`* is not modified and one of the errors below will be raised.

The port mode can not be set when the display hardware is detached due to the fact that there will be no valid port modes.

ERRORS

`WFD_ERROR_ILLEGAL_ARGUMENT`
- if *`mode`* is not a valid port mode for *`port`*

`WFD_ERROR_BAD_HANDLE`
- if *`port`* is not a valid `WFDPort` for *`device`*

`WFD_ERROR_NOT_SUPPORTED`
- if *`port`* is detached

4.4.5 Getting the currently set Port Mode

The most recently set port mode for a port can be retrieved by calling:

```
WFDPortMode wfdGetCurrentPortMode( WFDDevice device,
                                   WFDPort   port );
```

The *`device`* and *`port`* parameters denote the specific device and port, respectively, associated with this function call.

On success, the currently set port mode is returned via the function return value.

On failure, `WFD_INVALID_HANDLE` is returned and one of the errors below will be raised.

If a cached port mode exists, via a call to **wfdSetPortMode** without a following call to **wfdDeviceCommit** on *`port`*, the cached port mode will be returned by this function.

When a port is detached, see section 4.5.1.3, calling **wfdGetCurrentPortMode** will always result in a `WFD_ERROR_NOT_SUPPORTED` error since the port mode is not set.

ERRORS

`WFD_ERROR_NOT_SUPPORTED`
- if the port mode for *`port`* has not been set

`WFD_ERROR_BAD_HANDLE`
- if *`port`* is not a valid `WFDPort` for *`device`*

4.5 Port Configuration

For each `WFDPort` there are a number of attributes for status information and configuration. These attributes allow for the configuration of common display hardware and display port features. Many of these attributes are read-only, thus they only provide status information. Other attributes are read-writable and provide for user control.

Configuration attribute changes are not immediately applied to the hardware. They are cached until **wfdDeviceCommit** is called on the corresponding port. All retrieved attributes will reflect the cached values, if a cached value exists for the attribute.

Unless otherwise noted, the read-only attributes of the port configuration are static while display hardware is attached. See section 4.5.1.3 for attach/detach details. At the point when display hardware is attached or detached the read-only attributes will be updated as appropriate for the attached display hardware.

Several of the read-writable attributes are dependent on the port mode having been set, or cached, in order to properly range check attempted write values. Given this, the port mode must be set, or cached, prior to setting any configuration attributes. The range checking will be based on the cached port mode, if it exists. When there is no port mode set, or cached, the read-writable attributes can be read, but they will always return the default values.

4.5.1 Port Configuration Attributes

The `WFDPortConfigAttrib` enumeration lists the port configuration attributes.

```
typedef enum
{ WFD_PORT_ID                            = 0x7620,
  WFD_PORT_TYPE                          = 0x7621,
  WFD_PORT_DETACHABLE                    = 0x7622,
  WFD_PORT_ATTACHED                      = 0x7623,
  WFD_PORT_NATIVE_RESOLUTION             = 0x7624,
  WFD_PORT_PHYSICAL_SIZE                 = 0x7625,
  WFD_PORT_FILL_PORT_AREA                = 0x7626,
  WFD_PORT_BACKGROUND_COLOR              = 0x7627,
  WFD_PORT_FLIP                          = 0x7628,
  WFD_PORT_MIRROR                        = 0x7629,
  WFD_PORT_ROTATION                      = 0x762A,
  WFD_PORT_POWER_MODE                    = 0x762B,
  WFD_PORT_GAMMA_RANGE                   = 0x762C,
  WFD_PORT_GAMMA                         = 0x762D,
  WFD_PORT_PARTIAL_REFRESH_SUPPORT       = 0x762E,
  WFD_PORT_PARTIAL_REFRESH_MAXIMUM       = 0x762F,
  WFD_PORT_PARTIAL_REFRESH_ENABLE        = 0x7630,
  WFD_PORT_PARTIAL_REFRESH_RECTANGLE     = 0x7631,
  WFD_PORT_PIPELINE_ID_COUNT             = 0x7632,
  WFD_PORT_BINDABLE_PIPELINE_IDS         = 0x7633,
  WFD_PORT_PROTECTION_ENABLE             = 0x7634
} WFDPortConfigAttrib;
```

The storage type, default value, and read-writable state for all port configuration attributes are listed in ***Table 5***.

Attribute	Storage Type	R/W	Default
WFD_PORT_ID	WFDint	R	
WFD_PORT_TYPE	WFDPortType	R	
WFD_PORT_DETACHABLE	WFDboolean	R	
WFD_PORT_ATTACHED	WFDboolean	R	
WFD_PORT_NATIVE_RESOLUTION	WFDint[2]	R	
WFD_PORT_PHYSICAL_SIZE	WFDfloat[2]	R	
WFD_PORT_FILL_PORT_AREA	WFDboolean	R	
WFD_PORT_BACKGROUND_COLOR	WFDfloat[3]	R/W	(0,0,0)
WFD_PORT_FLIP	WFDboolean	R/W	WFD_FALSE
WFD_PORT_MIRROR	WFDboolean	R/W	WFD_FALSE
WFD_PORT_ROTATION	WFDint	R/W	0
WFD_PORT_POWER_MODE	WFDPowerMode	R/W	OFF
WFD_PORT_GAMMA_RANGE	WFDfloat[2]	R	
WFD_PORT_GAMMA	WFDfloat	R/W	1.0
WFD_PORT_PARTIAL_REFRESH_SUPPORT	WFDPartialRefresh	R	
WFD_PORT_PARTIAL_REFRESH_MAXIMUM	WFDint[2]	R	
WFD_PORT_PARTIAL_REFRESH_ENABLE	WFDPartialRefresh	R/W	NONE
WFD_PORT_PARTIAL_REFRESH_RECTANGLE	WFDint[4]	R/W	(0,0,0,0)
WFD_PORT_PIPELINE_ID_COUNT	WFDint	R	
WFD_PORT_BINDABLE_PIPELINE_IDS	WFDint[]	R	
WFD_PORT_PROTECTION_ENABLE	WFDboolean	R/W	WFD_FALSE

Table 5: Port Configuration Attributes

4.5.1.1 Port ID

The `WFD_PORT_ID` attribute is the ID of the port. This is the same value as retrieved from **wfdEnumeratePorts**.

Accessors: **wfdGetPortAttribi**

4.5.1.2 Port Type

The `WFD_PORT_TYPE` attribute denotes the type of display port hardware this `WFDPort` represents. The `WFDPortType` enumeration denotes the possible values.

```
typedef enum
{ WFD_PORT_TYPE_INTERNAL        = 0x7660,
  WFD_PORT_TYPE_COMPOSITE       = 0x7661,
  WFD_PORT_TYPE_SVIDEO          = 0x7662,
  WFD_PORT_TYPE_COMPONENT_YPbPr = 0x7663,
  WFD_PORT_TYPE_COMPONENT_RGB   = 0x7664,
  WFD_PORT_TYPE_COMPONENT_RGBHV = 0x7665,
  WFD_PORT_TYPE_DVI             = 0x7666,
  WFD_PORT_TYPE_HDMI            = 0x7667,
  WFD_PORT_TYPE_DISPLAYPORT     = 0x7668,
  WFD_PORT_TYPE_OTHER           = 0x7669
} WFDPortType;
```

The `WFD_PORT_TYPE_INTERNAL` value denotes hardwired internal display hardware.

The `WFD_PORT_TYPE_OTHER` value denotes a port type not otherwise listed.

The remaining port types denote common industry standard display hardware connection methods. The "`COMPONENT_RGB`" type assumes a "sync on green" connection where as the "`COMPONENT_RGBHV`" assumes separate horizontal and vertical sync signals.

This attribute is for UI reference only, to allow for the display of different device icons for instance. The user should not use this attribute to infer any further information.

Accessors: **wfdGetPortAttribi**

4.5.1.3 Detachable

The `WFD_PORT_DETACHABLE` attribute denotes whether the port supports dynamic attachment and detachment of display hardware. A value of `WFD_TRUE` denotes that the port supports this feature. The current attachment state of the port can be read via the `WFD_PORT_ATTACHED` attribute, for which a value of `WFD_TRUE` implies the display hardware is attached and available for use. For ports which are not detachable, the `WFD_PORT_ATTACHED` attribute will always be `WFD_TRUE`.

At the point when display hardware is attached, the following will occur:

- The number of available port modes, and their contents, will be determined based on the attached display hardware
- The available Standardized Display Data will be updated (see section 4.7 for details)
- A port attach event will be generated (see section 3.6.5.3 for event details)

At the point when display hardware is detached the following will occur:

- The port mode will revert to being unset
- The number of available port modes will be zero
- The port mode cached setting is cleared
- The available Standardized Display Data will be cleared (see section 4.7 for details)
- A port detach event will be generated (see section 3.6.5.3 for event details)

Accessors: **wfdGetPortAttribi**

4.5.1.4 Native Resolution

The `WFD_PORT_NATIVE_RESOLUTION` attribute defines the display hardware native, or optimal, resolution in pixels. This attribute is in the form of a two integer array for which the first array element denotes the width and the second array element denotes the height. If no such optimal resolution exists, these values will be zero.

Accessors: **wfdGetPortAttribiv**

4.5.1.5 Physical Size

The `WFD_PORT_PHYSICAL_SIZE` attribute defines the physical size of the content rendering area of the display hardware. This attribute is in the form of a two float array for which the first array element denotes the width and the second array element denotes the height. These parameters are in units of millimeters. If no physical size information is available these values will be zero.

Accessors: **wfdGetPortAttribfv**

4.5.1.6 Fill Port Area

The `WFD_PORT_FILL_PORT_AREA` attribute denotes whether or not the combined pipeline output image is required to fill the entire port displayable area. When this attribute is `WFD_TRUE`, the entire port displayable area must be filled with by the combined pipeline output image or undefined behavior will result.

Accessors: **wfdGetPortAttribi**

4.5.1.7 Background Color

The `WFD_PORT_BACKGROUND_COLOR` attribute defines the color that will contribute to areas not covered by opaque pipeline graphics data. This attribute is a three `WFDfloat` vector of the form (*`red`*, *`green`*, *`blue`*). This represents a non-premultiplied linear RGB color value in which each of the three colors has a range of 0 to 1.

This attribute has no effect if the `WFD_PORT_FILL_PORT_AREA` attribute is `WFD_TRUE`.

If accessed via the **wfdSet/GetPortAttribi** attribute functions, the attribute 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. When set, each color channel value is conceptually divided by 255.0f to obtain a value between 0 and 1. When retrieved, each color channel is clamped to the [0, 1] range, multiplied by 255 and rounded to obtain an 8-bit integer. The lowest 8 bits of the value must be 255 when written and will always read back as 255.

If accessed via the **wfdSet/GetPortAttribiv** attribute functions, the attribute is interpreted as an array of three `WFDuint8` vector of the form (*`red, green, blue`*). When set, each color channel value is conceptually divided by 255.0f to obtain a value between 0 and 1. When retrieved, each color channel is clamped to the [0, 1] range, multiplied by 255 and rounded to obtain an 8-bit integer.

Accessors: **wfdGetPortAttribi, wfdGetPortAttribiv, wfdGetPortAttribfv, wfdSetPortAttribi, wfdSetPortAttribiv, wfdSetPortAttribfv**

4.5.1.8 Flip

The `WFD_PORT_FLIP` attribute denotes whether flipping (inverting the displayed image top-to-bottom) is enabled or not. If this attribute is `WFD_TRUE` then flip is enabled. Flipping support is port mode dependent, see the `WFD_PORT_MODE_FLIP_MIRROR_SUPPORT` port mode attribute in section 4.4.1.3. Enabling and disabling the flip setting has no affect on other port or port mode attributes.

Accessors: **wfdGetPortAttribi**, **wfdSetPortAttribi**

4.5.1.9 Mirror

The `WFD_PORT_MIRROR` attribute denotes whether mirroring (inverting the displayed image left-to-right) is enabled or not. If this attribute is `WFD_TRUE` then mirroring is enabled. Mirroring support is port mode dependent, see the `WFD_PORT_MODE_FLIP_MIRROR_SUPPORT` port mode attribute in section 4.4.1.3. Enabling and disabling the mirror setting has no affect on other port or port mode attributes.

Accessors: **wfdGetPortAttribi**, **wfdSetPortAttribi**

4.5.1.10 Rotation

The `WFD_PORT_ROTATION` attribute denotes the current rotation. This value is in degrees of clockwise rotation, thus for a 90 degree clockwise rotation this attribute is set to 90. The `WFD_PORT_MODE_ROTATION_SUPPORT` port mode attribute denotes which rotation values are supported for the current mode, see section 4.4.1.4. Setting the rotation attribute no affect on other port or port mode attributes.

Rotation is applied after flip & mirror.

Accessors: **wfdGetPortAttribi**, **wfdSetPortAttribi**

4.5.1.11 Power Mode

The `WFD_PORT_POWER_MODE` attribute can be used to set the desired power mode of the port and associated display hardware. The `WFDPowerMode` enumeration defines the possible values for this attribute.

```
typedef enum
{ WFD_POWER_MODE_OFF             = 0x7680,
  WFD_POWER_MODE_SUSPEND         = 0x7681,
  WFD_POWER_MODE_LIMITED_USE     = 0x7682,
  WFD_POWER_MODE_ON              = 0x7683
} WFDPowerMode;
```

With some types of display ports the platform may have little or no control over the actual display hardware power state, for instance an S-Video port. Due to this limitation the power state descriptions below define the display hardware power state in terms of the ideal state, assuming control over the display hardware.

The `WFD_POWER_MODE_OFF` value denotes the fully off state for the port and the display hardware. This state is defined as follows:

- The display port will be disabled and the power usage of the port will be at its minimum.
- The display hardware will be at the lowest power consumption state controllable by the platform, ideally fully off.
- No image data will be rendered to the display hardware and the most recently rendered image will be lost by the port.
- The time it takes to return to rendering image data to the display hardware is the longest of all the power states

The `WFD_POWER_MODE_SUSPEND` value denotes a mostly but not completely off state. This state is targeted towards platforms with display ports or controllable display hardware that have long turn on times. This state can be defined as follows:

- The display port will be disabled and the power usage of the port can be greater than that of the "Off" state.
- The display hardware will be ideally in a low power consumption state.

- No image data will be rendered to the display hardware and the most recently rendered image will be lost by the port.
- The time it takes to return to rendering image data to the display hardware should be shorter than the "Off" state.

The `WFD_POWER_MODE_LIMITED_USE` value denotes a mostly but not completely on state. This state is targeted towards platforms with smart display hardware that does not require constant refreshing, thus can reduce power consumption when the displayed content is not rapidly being updated. This state can be defined as follows:

- The display port will be in a lower power consumption mode than the "On" state.
- The display hardware will be on.
- Image data will be maintained by the display hardware.

The `WFD_POWER_MODE_ON` value denotes the fully on state for the port and the display hardware, thus ready for continuous use.

Table 6 consolidates the above information into a shortened form:

State	Visible Content	Recovery Time	Refresh Rate
On	Y	N/A	Port Refresh Rate
Limited Use	Y	N/A	Less than "On"
Suspend	N	Less than "Off"	N/A
Off	N	Full Initialization	N/A

Table 6: Power Modes

Accessors: **wfdGetPortAttribi**, **wfdSetPortAttribi**

4.5.1.12 Gamma

Gamma correction is applied to each pixel of the output image before it is sent to the display hardware. Gamma correction is applied per color component (red, green, blue) of each pixel as defined by:

$$Output_X = Max_X * ((Input_X / Max_X) \wedge Gamma)$$

Where: $Input_X$ and $Output_X$ are the input and output pixel color component values, respectively; Max_X is the maximum value for the color component; the "^" implies exponentiation; and *Gamma* is the client provided gamma correction value, set via the `WFD_PORT_GAMMA` attribute.

The `WFD_PORT_GAMMA_RANGE` attribute defines the range of valid values for `WFD_PORT_GAMMA`. This attribute is in the form of a two floating-point value array for which the first array element denotes the minimum gamma value and the second array element denotes the maximum gamma value. In cases where client specified gamma correction is not supported, the minimum and maximum values will be equal.

Gamma correction is only applied to RGB color spaces.

Accessors (`GAMMA_RANGE`): **wfdGetPortAttribfv**
Accessors (`GAMMA`): **wfdGetPortAttribf**, **wfdSetPortAttribf**

4.5.1.13 Partial Refresh

The `WFD_PORT_PARTIAL_REFRESH_SUPPORT` attribute denotes which partial refresh modes are supported, if any. The `WFDPartialRefresh` enumeration defines the possible support.

```
typedef enum
{ WFD_PARTIAL_REFRESH_NONE       = 0x7690,
  WFD_PARTIAL_REFRESH_VERTICAL   = 0x7691,
  WFD_PARTIAL_REFRESH_HORIZONTAL = 0x7692,
  WFD_PARTIAL_REFRESH_BOTH       = 0x7693
} WFDPartialRefresh;
```

The `WFD_PORT_PARTIAL_REFRESH_MAXIMUM` attribute denotes the maximum width and height of the partial refresh rectangle. This attribute is a two integer vector of the form (*`width, height`*).

The `WFD_PORT_PARTIAL_REFRESH_ENABLE` attribute is used to set an active partial refresh mode. The `WFDPartialRefresh` enumeration also defines the possible enabled modes. Attempting to enable a non-supported mode will result in a `WFD_ERROR_ILLEGAL_ARGUMENT`.

The `WFD_PORT_PARTIAL_REFRESH_RECT` attribute is used to define the refresh rectangle. This attribute is a four integer vector of the form (*`offsetX, offsetY, width, height`*).

When vertical partial refresh mode is enabled *`offsetY`* and *`height`* are used. *`offsetY`* is used to define the pixel offset from the top edge of the display to the top edge of the refresh area. A value of 0 denotes that the top edge of the port is within the refresh area. *`height`* is used to define the pixel offset to the bottom edge of the refresh area. The bottom edge of the refresh area is defined as *`offsetY`* + *`height`*. The bottom edge of the refresh area must be no greater than the full display height in pixels. A *height* value of 0 results in the same behavior as disabling vertical partial refresh mode. A *height* value greater than the WFD_PORT_PARTIAL_REFRESH_MAXIMUM height value will result in a WFD_ERROR_INVALID_ARGUMENT error.

When horizontal partial refresh mode is enabled *offsetX* and *width* are used. *offsetX* is used to define the pixel offset from the left edge of the display to the left edge of the refresh area. A value of 0 denotes that the left edge of the display is within the refresh area. *width* is used to define the pixel offset to the right edge of the refresh area. The right edge of the refresh area is defined as *offsetX* + *width*. The right edge of the refresh area must be no greater than the full display width in pixels. A *width* value of 0 results in the same behavior as disabling horizontal partial refresh mode. A *width* value greater than the WFD_PORT_PARTIAL_REFRESH_MAXIMUM width value will result in a WFD_ERROR_INVALID_ARGUMENT error.

Accessors (SUPPORT): **wfdGetPortAttribi**
Accessors (MAXIMUM): **wfdGetPortAttribiv**
Accessors (ENABLE): **wfdGetPortAttribi, wfdSetPortAttribi**
Accessors (RECTANGLE): **wfdGetPortAttribiv, wfdSetPortAttribiv**

4.5.1.14 Bindable Pipeline IDs

The port configuration contains a list of pipelines, via pipeline ID, which can be bound to the port, see section 4.6. This list consists of two attributes: WFD_PORT_PIPELINE_ID_COUNT which denotes the number of pipeline IDs that are in the list; and WFD_PORT_BINDABLE_PIPELINE_IDS which contains the list and is in the form of a vector of WFDint values.

It is possible for a port to have no bindable pipelines, in which case no pipeline ID will be present in this list and WFD_PORT_PIPELINE_ID_COUNT will be zero.

Accessors (PIPELINE_ID_COUNT): **wfdGetPortAttribi**
Accessors (BINDABLE_PIPELINE_IDS): **wfdGetPortAttribiv**

4.5.1.15 Content Protection

The WFD_PORT_PROTECTION_ENABLE attribute denotes the enabling (WFD_TRUE) or disabling (WFD_FALSE) of port content protection. When content protection is successfully enabled, the image data delivered to the port will be transferred to the display hardware in a secure manner.

The WFD_ERROR_INVALID_ARGUMENT error will occur when attempting to enable content protection, if the display control hardware, attached display hardware, or the connection interface between them does not support content protection.

If at any time, while content protection is enabled, the secure transfer means is determined to be no longer secure, the following will occur:

- Any existing pipeline binding will be forcibly unbound.
- An event will occur. See section 3.6.5.4 for details.
- Attempts to bind a pipeline to the port will fail until either content protection is disabled or the secure transfer means is determined to be secure again. The error generated for this condition will be `WFD_ERROR_NOT_SUPPORTED`.

Accessors: **wfdGetPortAttribi**, **wfdSetPortAttribi**

4.5.2 Querying Port Configuration Attributes

A port configuration attribute can be queried by calling one of the following functions:

```
WFDint wfdGetPortAttribi( WFDDevice          device,
                          WFDPort            port,
                          WFDPortConfigAttrib attrib );

WFDfloat wfdGetPortAttribf( WFDDevice          device,
                            WFDPort            port,
                            WFDPortConfigAttrib attrib );

void wfdGetPortAttribiv( WFDDevice          device,
                         WFDPort            port,
                         WFDPortConfigAttrib attrib,
                         WFDint             count,
                         WFDint            *value );

void wfdGetPortAttribfv( WFDDevice          device,
                         WFDPort            port,
                         WFDPortConfigAttrib attrib,
                         WFDint             count,
                         WFDfloat          *value );
```

The *`device`* and *`port`* parameters denote the specific device and port, respectively, associated with this function call.

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 *`value`* must be an array of at least *`count`* elements.

On success, the value(s) of *`attrib`* for port is returned via the function return value or the *`value`* parameter.

On failure, one of the errors below will be raised. In addition, for the vector based functions, *`value`* is not modified.

If cached attributes exists, via a call to **wfdSetPortAttribx** without a following call to **wfdDeviceCommit** on *`port`*, the cached attributes will be returned by this function.

ERRORS

`WFD_ERROR_ILLEGAL_ARGUMENT`
- if *`value`* is `NULL`
- if *`count`* does not match the attribute element count

`WFD_ERROR_BAD_HANDLE`
- if *`port`* is not a valid `WFDPort` for *`device`*

`WFD_ERROR_BAD_ATTRIBUTE`
- if *`attrib`* is not a valid port configuration attribute
- if *`attrib`* does not permit the use of the accessor

4.5.3 Setting Port Configuration Attributes

A port configuration attribute can be set by calling one of the following functions:

```
void wfdSetPortAttribi( WFDDevice           device,
                        WFDPort             port,
                        WFDPortConfigAttrib attrib,
                        WFDint              value );

void wfdSetPortAttribf( WFDDevice           device,
                        WFDPort             port,
                        WFDPortConfigAttrib attrib,
                        WFDfloat            value );

void wfdSetPortAttribiv( WFDDevice           device,
                         WFDPort             port,
                         WFDPortConfigAttrib attrib,
                         WFDint              count,
                         const WFDint       *value );

void wfdSetPortAttribfv( WFDDevice           device,
                         WFDPort             port,
                         WFDPortConfigAttrib attrib,
                         WFDint              count,
                         const WFDfloat     *value );
```

The *`device`* and *`port`* parameters denote the specific device and port, respectively, associated with this function call.

The *`attrib`* parameter denotes the attribute to set to *`value`*. For the vector based functions, *`count`* denotes the number of values provided and *`value`* must be an array of at least *`count`* elements.

On success, the setting of *`value`* to *`attrib`* for *`port`* will be cached. The changes will be applied to the hardware when **wfdDeviceCommit** is called on *`port`*.

On failure, port configuration attribute is not modified and one of the errors below will be raised.

ERRORS

`WFD_ERROR_ILLEGAL_ARGUMENT`
- if *`value`* is `NULL`
- if *`value`* is invalid or out-of-range
- if *`count`* does not match the attribute element count

`WFD_ERROR_NOT_SUPPORTED`
- if the mode of *`port`* has not been set

`WFD_ERROR_BAD_HANDLE`
- if *`port`* is not a valid `WFDPort` for *`device`*

`WFD_ERROR_BAD_ATTRIBUTE`
- if *`attrib`* is not a valid port configuration attribute
- if *`attrib`* denotes a read-only attribute
- if *`attrib`* does not permit the use of the accessor

4.6 Binding a Pipeline to a Port

In order to provide an output image to the display hardware attached to a port, one or more pipelines must be bound to the port to provide the image data. The pipelines that can be bound to a port are listed via port configuration attributes, see section 4.5.1.14 for details.

A pipeline is bound to a port by calling:

```
void wfdBindPipelineToPort( WFDDevice   device,
                            WFDPort     port,
                            WFDPipeline pipeline );
```

The *`device`* and *`port`* parameters denote the specific device and port, respectively, associated with this function call.

The *`pipeline`* parameter is the handle to the pipeline to bind to *`port`*.

On success, the binding of *`pipeline`* to *`port`* will be cached. The binding will be completed when **wfdDeviceCommit** is called on *`port`*. If *`pipeline`* is currently bound to another port, that binding will be released when **wfdDeviceCommit** is called, but only if it is called on *`device`* since a second port is involved.

On failure, neither *`port`* nor *`pipeline`* are modified and one of the below errors will be generated.

ERRORS

`WFD_ERROR_NOT_SUPPORTED`
- if content protection is enabled and *`port`* is not secure

`WFD_ERROR_BAD_HANDLE`
- if *`port`* is not a valid `WFDPort` for *`device`*
- if *`pipeline`* is not a valid `WFDPipeline` for *`device`* or *`port`*

4.7 Standardized Display Data

Some display hardware supports the retrieval of one or more standardized lists of information about the hardware via VESA Display Data Channel or some other forms of information exchange. These lists of information can exist in several different standardized forms. The `WFDDisplayDataFormat` enumeration lists the possible formats of this display information.

```
typedef enum
{ WFD_DISPLAY_DATA_FORMAT_NONE      = 0x76A0,
  WFD_DISPLAY_DATA_FORMAT_EDID_V1   = 0x76A1,
  WFD_DISPLAY_DATA_FORMAT_EDID_V2   = 0x76A2,
  WFD_DISPLAY_DATA_FORMAT_DISPLAYID = 0x76A3
} WFDDisplayDataFormat;
```

4.7.1 Supported Display Data Formats

A list of display data formats supported by a given port can be retrieved by calling:

```
WFDint wfdGetDisplayDataFormats
                     ( WFDDevice            device,
                       WFDPort              port,
                       WFDDisplayDataFormat *format,
                       WFDint               formatCount );
```

The *`device`* and *`port`* parameters denote the specific device and port, respectively, associated with this function call.

The *`format`* and *`formatCount`* parameters define the caller provided array to return the supported formats into.

On success, the number of *`format`* elements that were populated with supported display data format values is returned via the function return value. Thus elements 0 through (the returned value) - 1 of *`format`* will contain valid display data format values. No more than *`formatCount`* values will be returned, even if more are available. However, if **wfdGetDisplayDataFormats** is called with *`format`* = `NULL` then no display data format values are returned, but the total number of supported display data format values is returned via the function return value.

On failure, zero is returned and *`format`* is not modified. Additionally, one of the errors below will be raised.

ERRORS

`WFD_ERROR_ILLEGAL_ARGUMENT`
- if *`formatCount`* is invalid, zero or negative

`WFD_ERROR_BAD_HANDLE`
- if *`port`* is not a valid `WFDPort` for *`device`*

4.7.2 Retrieving Display Data

The actual data bytes from a standardized list of display data can be retrieved by calling:

```
WFDint wfdGetDisplayData( WFDDevice            device,
                          WFDPort              port,
                          WFDDisplayDataFormat format,
                          WFDuint8            *data,
                          WFDint               dataCount );
```

The *`device`* and *`port`* parameters denote the specific device and port, respectively, associated with this function call.

The *`format`* parameter denotes the desired display data list to retrieve.

The *`data`* and *`dataCount`* parameters define the caller provided array to return the standardized list of data into.

On success, the number of bytes of display data information written to *`data`* is returned via the function return value. Thus bytes 0 through (the returned value) - 1 of *`data`* will contain valid bytes from the display data information list for *`format`*. The display data information is always returned as consecutive bytes with the first byte of the list returned in byte 0 of *`data`*. No more than *`dataCount`* values will be returned, even if more are available. However, if **wfdGetDisplayData** is called with *`data`* = `NULL` then no display data bytes are returned, but the total number of bytes in the display data information list for *`format`* is returned via the function return value.

On failure, zero is returned and *`format`* is not modified. Additionally, one of the errors below will be raised.

ERRORS

`WFD_ERROR_ILLEGAL_ARGUMENT`
- if *`format`* is not a valid display data format for *`port`*
- if *`dataCount`* is invalid, zero or negative

`WFD_ERROR_BAD_HANDLE`
- if *`port`* is not a valid `WFDPort` for *`device`*

5 Pipelines

The content that is rendered to display hardware is delivered to the display control device via a hardware graphics pipeline. A `WFDPipeline`, which is `WFDDevice` specific, is an abstraction of a hardware graphics pipeline. These pipelines provide a means for configuring the way in which graphics data is composed together and finally output to a display port and onto display hardware.

A single `WFDDevice` must support zero or more pipelines[8].

5.1 Pipeline Stages

The following diagram depicts the stages of a display control pipeline:

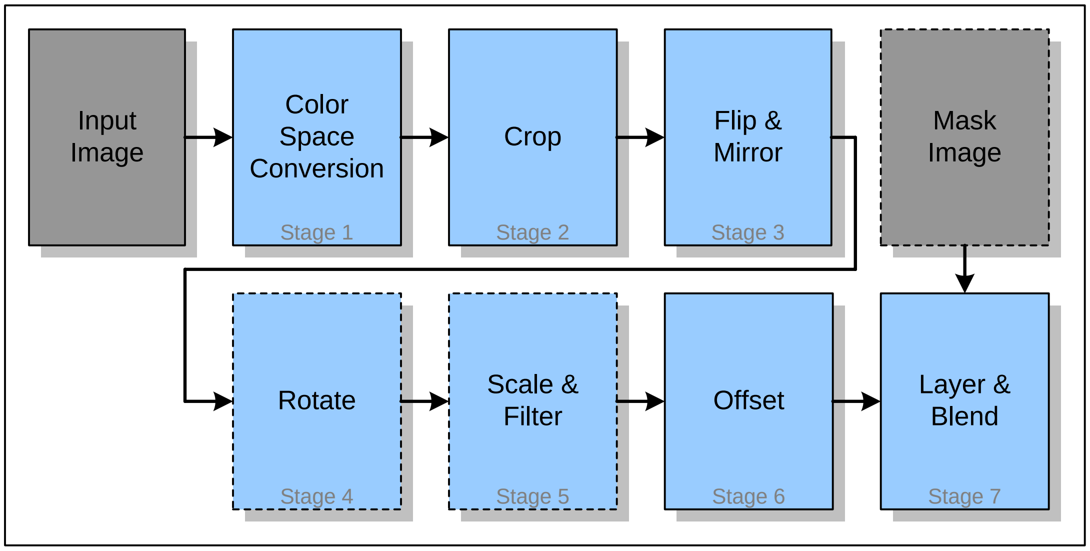

Figure 2: Display Control Pipeline Stages

Implementations are not required to match the ideal pipeline stage-for-stage. They may take any approach to rendering so long as the final results match the results of the ideal pipeline.

5.1.1 Stage 1 – Color Space Conversion

During the color space conversion stage, the input image will be converted to the pixel format necessary for generating the proper output image for the desired display port. This operation is done without any direct user input, but is shown for completeness.

[8] The expected case for zero exposed pipelines is if the same manufacturer provides the OpenWF Display implementation as well as other graphics composition APIs, such that exposing the OpenWF Display pipelines would be unnecessary since other more capable, less limited, composition graphics APIs exist.

5.1.2 Stage 2 – Crop

During the crop stage, the input image can be cropped to a sub-rectangle of the input image. The user defines this sub-rectangle, the source rectangle, via an X offset, Y offset, width, and height. If no cropping is desired, the source rectangle must be defined with the X & Y offsets set to zero and the width & height matching the input image width & height.

5.1.3 Stage 3 – Flip & Mirror

During the flip & mirror stage, the crop output can be flipped (inverting the image top-to-bottom) and/or mirrored (inverting the image left-to-right). The user defines if this operation is performed.

5.1.4 Stage 4 – Rotate

During the rotate stage, the flip & mirror output can be rotated clockwise. The user defines the degrees of rotation.

5.1.5 Stage 5 – Scale & Filter

During the scale and filter stage, the rotate output can be scaled up or scaled down in size and a scaling filter can be applied. The user defines the amount of scaling via the destination rectangle width & height and the user defines the type of filtering desired, if any.

5.1.6 Stage 6 – Offset

During the offset stage, the scale & filter output will be placed in the viewable region of the output. The user defines the placement via the destination rectangle X and Y offsets that are define in reference to the associated display port width and height.

5.1.7 Stage 7 – Layer & Blend

During the final stage, the layer and blend stage, the resultant image from all the previous stages will be combined with output from all of the other pipelines feeding into the associated display port. The layering occurs based on the hardware definition, as indicated in the pipeline configuration layer attribute. The blending occurs based on pipeline configuration transparency settings and the mask image, if attached.

5.2 Enumerating Pipelines

The number and IDs of the available pipelines for a given device can be retrieved by calling:

```
WFDint wfdEnumeratePipelines
                          ( WFDDevice     device,
                            WFDint       *pipelineIds,
                            WFDint        pipelineIdsCount
                            const WFDint *filterList );
```

The *device* parameter denotes the specific device to enumerate the pipelines for.

The *pipelineIds* and *pipelineIdsCount* parameters define the caller provided array to return the pipeline IDs into.

The *filterList* parameter contains a list of filtering attributes which are used to control the pipeline IDs returned by **wfdEnumeratePipelines**. All attributes in *filterList* are immediately followed by the corresponding value. The list is terminated with `WFD_NONE`. *filterList* may be `NULL` or empty (first attribute is `WFD_NONE`), in which case all valid pipeline IDs for the platform are returned. Providing a non-`NULL` *filterList* that is not terminated with `WFD_NONE` will result in undefined behavior. Providing an invalid filter attribute or filter attribute value will result in no pipeline IDs being returned and a return value of zero. No valid filtering attributes are defined for *filterList* in this specification.

On success, the number of *pipelineIds* elements that were populated with pipeline ID values is returned via the function return value. Thus elements 0 through (the returned value) - 1 of *pipelineIds* will contain valid pipeline ID values. No more than *pipelineIdsCount* values will be returned, even if more are available. However, if **wfdEnumeratePipelines** is called with *pipelineIds* = `NULL` then no pipeline IDs are returned, but the total number of available pipeline IDs is returned via the function return value.

On failure, zero is returned and *pipelineIds* is not modified. Additionally, one of the errors below will be raised.

All pipeline IDs on a hardware platform are unique, but should not be expected to be contiguous. A pipeline ID equal to `WFD_INVALID_PIPELINE_ID` will not be returned.

```
#define WFD_INVALID_PIPELINE_ID (0)
```

The list of available pipelines is static for as long as the associated `WFDDevice` handle is valid.

ERRORS

`WFD_ERROR_ILLEGAL_ARGUMENT`
- if *`pipelineIdsCount`* is invalid, zero or negative

5.3 Creating a Pipeline

A pipeline can be created by calling:

```
WFDPipeline wfdCreatePipeline( WFDDevice     device,
                               WFDint        pipelineId,
                               const WFDint *attribList );
```

The *`device`* parameter denotes the specific device associated with this function call.

The *`pipelineId`* parameter denotes the graphics pipeline to create. This value should be a pipeline ID retrieved using the **wfdEnumeratePipelines** function.

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

On success, a `WFDPipeline` representing *`pipelineId`* is created and a valid handle is returned.

On failure, `WFD_INVALID_HANDLE` is returned and one of the following errors will be raised.

Only one outstanding instance of a `WFDPipeline` is allowed for any specific pipeline. While an instance of a specific pipeline is outstanding, created but not yet destroyed, any further attempts to create the same pipeline will fail, resulting in a `WFD_ERROR_IN_USE` error and `WFD_INVALID_HANDLE` being returned.

ERRORS

`WFD_ERROR_ILLEGAL_ARGUMENT`
- if *`pipelineId`* is not a valid pipeline ID for `device`

`WFD_ERROR_BAD_ATTRIBUTE`
- if *`attribList`* contains invalid attributes or invalid attribute values

`WFD_ERROR_OUT_OF_MEMORY`
- if lack of sufficient memory prevents the creation of the `WFDPipeline`

`WFD_ERROR_IN_USE`
- if a `WFDPipeline` associated with *`pipelineId`* has already been created

`WFD_ERROR_BUSY`
- if the pipeline associated with *`pipelineId`* is in use indirectly and can not immediately be released for direct use.

5.4 Destroying a Pipeline

A graphics pipeline is destroyed by calling:

```
void wfdDestroyPipeline( WFDDevice   device,
                         WFDPipeline pipeline );
```

The *`device`* parameter denotes the specific device associated with this function call.

The *`pipeline`* parameter denotes the pipeline to destroy. Following this call, all resources associated with *`pipeline`* are marked for deletion as soon as possible. The pipeline handle, *`pipeline`*, is no longer valid. All resource handles associated with *`pipeline`* become invalid. Any references held by *`pipeline`* on external resources are removed.

All pending usages of input resources associated with *`pipeline`* are completed before this call completes.

A destroyed pipeline can be re-created given the associated pipeline ID and a call to **wfdCreatePipeline**. The handle to a re-created pipeline should not be expected to match the previously destroyed handle.

ERRORS

`WFD_ERROR_BAD_HANDLE`
- if *`pipeline`* is not a valid `WFDPipeline` for *`device`*

5.5 Image Providers

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

An image provider can be created from an image or a stream. The act of deriving an image provider from an image or stream, as well as the subsequent usage of the image provider within OpenWF Display, will never modify the pixel data of the image or stream[9]. Support for discovering image or stream formats compatible with a Device or Pipeline is outside of the scope of OpenWF Display.

5.5.1 Sources

A Source is a supplier of image data that a pipeline 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.

5.5.1.1 Creating Sources

A source is created by calling:

```
WFDSource wfdCreateSourceFromImage
                      ( WFDDevice     device,
                        WFDPipeline   pipeline,
                        WFDEGLImage   image,
                        const WFDint *attribList );

WFDSource wfdCreateSourceFromStream
                      ( WFDDevice            device,
                        WFDPipeline          pipeline,
                        WFDNativeStreamType  stream,
                        const WFDint        *attribList );
```

The *device* and *pipeline* parameters denote the specific device and pipeline associated with this function call.

The *image* parameter must denote a valid `WFDEGLImage` which must be compatible with *device* and *pipeline*.

[9] 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. OpenWF Display relies on the image or stream being allocated in a format and location suitable for OpenWF Display at creation time.

The *`stream`* parameter must denote a valid stream which must be compatible with *`device`* and *`pipeline`*.

No valid attributes are defined for *`attribList`* in this specification.

On success, a `WFDSource` representing *`image`* or *`stream`* is created and a valid handle is returned. The returned `WFDSource` places a reference on *`image`* or *`stream`*. For image based sources, this reference is a sibling reference.

On failure, `WFD_INVALID_HANDLE` is returned and one of the following errors will be raised.

If multiple Sources are created from the same stream, each Source will have a unique handle.

ERRORS

`WFD_ERROR_OUT_OF_MEMORY`
- if the implementation fails to allocate resources for the source

`WFD_ERROR_ILLEGAL_ARGUMENT`
- if *`image`* is not a valid `WFDEGLImage`
- if *`stream`* is not a valid `WFDNativeStreamType`

`WFD_ERROR_NOT_SUPPORTED`
- if *`image`* or *`stream`* is valid but not suitable for use by *`device`* or *`pipeline`*

`WFD_ERROR_BUSY`
- if *`stream`* can not be used at this time due to other usages of *`stream`*

`WFD_ERROR_BAD_HANDLE`
- if *`pipeline`* is not a valid `WFDPipeline` for *`device`*

5.5.1.2 Destroying Sources

A source is destroyed by calling:

```
void wfdDestroySource( WFDDevice device
                       WFDSource source );
```

The *`device`* parameter denotes the specific device associated with this function call.

The *`source`* parameter denotes the source to destroy.

Following this call, all resources associated with *`source`* which were allocated by the implementation are marked for deletion as soon as possible. The reference placed on the source's image or stream at creation time is removed. *`source`* is no longer a valid source handle. Any references held by *`source`* on external resources are removed.

ERRORS

`WFD_ERROR_BAD_HANDLE`
- if *`source`* is not a valid `WFDSource` for *`device`*

5.5.2 Masks

A Mask is a supplier of image data that a pipeline 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.

5.5.2.1 Creating Masks

A mask is created by calling:

```
WFDMask wfdCreateMaskFromImage
                      ( WFDDevice     device,
                        WFDPipeline   pipeline,
                        WFDEGLImage   image,
                        const WFDint *attribList );

WFDMask wfdCreateMaskFromStream
                      ( WFDDevice            device,
                        WFDPipeline          pipeline,
                        WFDNativeStreamType  stream,
                        const WFDint        *attribList );
```

The *`device`* and *`pipeline`* parameters denote the specific device and pipeline associated with this function call.

The *`image`* parameter must denote a valid `EGLImage` which must be compatible with *`device`* and *`pipeline`*.

The *`stream`* parameter must denote a valid stream which must be compatible with *`device`* and *`pipeline`*.

No valid attributes are defined for *`attribList`* in this specification.

On success, a `WFDMask` representing *`image`* or *`stream`* is created and a valid handle is returned. The returned `WFDMask` places a reference on *`image`* or *`stream`*. For image based masks, this reference is a sibling reference.

On failure, `WFD_INVALID_HANDLE` is returned and one of the following errors will be raised.

If multiple Masks are created from the same stream, each Mask will have a unique handle.

ERRORS

`WFD_ERROR_OUT_OF_MEMORY`
- if the implementation fails to allocate resources for the mask

`WFD_ERROR_ILLEGAL_ARGUMENT`
- if *`image`* is not a valid `WFDEGLImage`
- if *`stream`* is not a valid `WFDNativeStreamType`

`WFD_ERROR_NOT_SUPPORTED`
- if *`image`* or *`stream`* is valid but not suitable for use by *`device`* or *`pipeline`*

`WFD_ERROR_BUSY`
- if *`stream`* can not be used at this time due to other usages of *`stream`*

`WFD_ERROR_BAD_HANDLE`
- if *`pipeline`* is not a valid `WFDPipeline` for *`device`*

5.5.2.2 Destroying Masks

A mask is destroyed by calling:

```
void wfdDestroyMask( WFDDevice device,
                     WFDMask   mask );
```

The *`device`* parameter denotes the specific device associated with this function call.

The *`mask`* parameter denotes the mask to destroy.

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 image or stream at creation time is removed. *`mask`* is no longer a valid mask handle. Any references held by *`mask`* on external resources are removed.

ERRORS

`WFD_ERROR_BAD_HANDLE`
- if *`mask`* is not a valid `WFDMask` for *`device`*

5.6 Using a Pipeline

There are two models of use of a pipeline for graphical content input, a direct content input model and an indirect content input model.

The direct content input model consist of the case where the OpenWF Display API user creates the pipeline handles for the desired pipeline(s), creates one or more source and/or mask handles with input content, and finally directly binding that content to the desired pipeline(s). In this model, the user is in complete control of all aspect of the composition being performed by the display controller hardware for the created pipeline(s). The act of binding a source to the pipeline is the means of enabling the pipeline for rendering. In order for the rendering to be disabled, the source binding must be released.

The indirect content input model is the case where the OpenWF Display API user performs no pipeline creations or configurations. In this case, an outside entity, such as OpenWF Composition, will make use of unused pipelines for composition as it sees fit. This is the default model for all pipelines.

The OpenWF Display API user can at anytime attempt to reserve one or more pipelines for direct use by creating the pipeline handle(s) to the desired pipeline(s). In some cases the pipeline creation may fail if the pipeline is in use indirectly and, for implementation specific reasons, can not be immediately freed. The creation of a pipeline handle implies the desire for direct use for that specific pipeline. Other pipelines that are not being directly used may still be used by another entity.

A pipeline that has been previously allocated, its handle created by the user, will be released back for indirect use when the pipeline handle is destroyed by the user.

5.6.1 Mixed Model Pipeline Usage Considerations

Since it is possible, but not encouraged, to operate some pipelines directly (direct model) while other pipelines are operating indirectly (indirect model), the user must pay special attention to this mixed use case.

The pipeline layering order is very important to consider when operating in a mixed use case. Pipelines should always be used in layering groups, such that a consecutive set of layers are operating in the same model. For example, assume there exists a port with three pipelines such that pipeline 1 is the lowest layer and pipeline 3 is the highest layer. A typical use cases would be to leave all layer operating indirectly (used by OpenWF Composition), or possibly leave layers 1 & 2 operating indirectly and use layer 3 directly for some special manually rendered content. A use case where a middle layer, layer 2, is used directly while the others remain available for indirect use could produce undefined effects.

Some users, such as OpenWF Composition, output completely filled and opaque content thus it only makes sense to provide a consecutive set of layers which includes the lowest layer. Thus, using the example above, leaving layers 1 & 2 to OpenWF Composition and using layer 3 for some other content would result in a potential proper overlaying effect. In contrast, leaving layers 2 & 3 to OpenWF Composition will always result in the content of layer 1 being completely obscured, thus never visible on the port output.

5.6.2 Binding a Source to a Pipeline

A source can be bound to a pipeline by calling:

```
void wfdBindSourceToPipeline( WFDDevice      device,
                              WFDPipeline    pipeline,
                              WFDSource      source,
                              WFDTransition  transition,
                              const WFDRect *region );
```

The `device` and `pipeline` parameters denote the specific device and graphics pipeline, respectively, associated with this function call.

The `source` parameters denote the image provider to be bound to `pipeline`. If a current binding to `pipeline` exists, the current binding will be released when the new binding occurs. If `source == WFD_INVALID_HANDLE` then no new source will be bound to `pipeline`, but any current binding will still be released.

The `transition` parameter denotes when the binding is to occur. See section 5.6.4 for details.

The `region` parameter is a `WFDRect` and denotes a rectangle of the image that has changed from the previous image. `region`'s relate to `EGLImage` sources only, for stream sources `region` must always be `NULL`. See section 8 concerning the proper use of this parameter. If this parameter is `NULL,` it is assumed that the entire image has changed.

```
typedef struct
{ WFDint offsetX;
  WFDint offsetY;
  WFDint width;
  WFDint height;
} WFDRect;
```

On success, the binding of *`source`* to *`pipeline`*, and the release of any current binding, will be applied as defined by *`transition`*. If *`source`* `==` `WFD_INVALID_HANDLE`, no source will be bound to *`pipeline`*, but any current binding will be released as defined by *`transition`*.

On failure, *`pipeline`* is not modified and one of the errors below will be raised.

ERRORS

`WFD_ERROR_ILLEGAL_ARGUMENT`
- if *`transition`* is not a valid `WFDTransition`
- if *`source`* is neither a valid `WFDSource` nor `WFD_INVALID_HANDLE`
- if *`source`* was not created for *`pipeline`*
- if *`region`* is non-NULL when *`source`* is a stream

`WFD_ERROR_BAD_HANDLE`
- if *`pipeline`* is not a valid `WFDPipeline` for *`device`*

5.6.3 Binding a Mask to a Pipeline

A `WFDMask` can be bound to a pipeline by calling:

```
void wfdBindMaskToPipeline( WFDDevice     device,
                            WFDPipeline   pipeline,
                            WFDMask       mask,
                            WFDTransition transition );
```

The *`device`* and *`pipeline`* parameters denote the specific device and graphics pipeline, respectively, associated with this function call.

The *`mask`* parameter denotes the imager provider to be bound to *`pipeline`*. If a current binding to *`pipeline`* exists, the current binding will be released when the new binding occurs. If *`mask`* `==` `WFD_INVALID_HANDLE` then no new mask will be bound to *`pipeline`*, but any current binding will still be released.

The *`transition`* parameter denotes when the binding is to occur. See section 5.6.4 for details.

On success, the binding of `mask` to `pipeline`, and the release of any current binding, will be applied as defined by `transition`. If `mask == WFD_INVALID_HANDLE`, no mask will be bound to `pipeline`, but any current binding will be released as defined by `transition`.

On failure, `pipeline` is not modified and one of the errors below will be raised.

If a Mask is set, it must be equal in size to the pipeline's destination rectangle (`WFD_PIPELINE_DESTINATION_RECTANGLE)` when **wfdDeviceCommit** is called. Otherwise the pipeline will be considered inconsistent. This restriction may be lifted in future revisions of this specification.

ERRORS

`WFD_ERROR_ILLEGAL_ARGUMENT`
- if `transition` is not a valid `WFDTransition`
- if `mask` is neither a valid `WFDMask` nor `WFD_INVALID_HANDLE`
- if `mask` was not created for `pipeline`

`WFD_ERROR_BAD_HANDLE`
- if `pipeline` is not a valid `WFDPipeline` for `device`

5.6.4 Image Transitions

The term "Image Transition" relates to how, or when, a new pipeline binding will take effect. These transitions occur in the following cases:

- a transition from a pipeline with no source bindings to one with a bound source, via **wfdBindSourceToPipeline**
- a transition replacing a currently bound source bindings to another source, via **wfdBindSourceToPipeline**
- a transition removing a currently bound source, via **wfdBindSourceToPipeline**
- a transition from a pipeline with no mask bindings to one with a bound mask, via **wfdBindMaskToPipeline**
- a transition replacing a currently bound mask bindings to another mask, via **wfdBindMaskToPipeline**
- a transition removing a currently bound mask, via **wfdBindMaskToPipeline**

For each of these cases the `WFDTransition` enumeration defines the possible transition timing.

```
typedef enum
{ WFD_TRANSITION_INVALID     = 0x77E0,
  WFD_TRANSITION_IMMEDIATE   = 0x77E1,
  WFD_TRANSITION_AT_VSYNC    = 0x77E2
} WFDTransition;
```

All transitions will occur during the next **wfdDeviceCommit** call affecting the associated pipeline. The `WFD_TRANSITION_IMMEDIATE` value will cause the transition to occur as soon as possible, without concern for VSYNC. The `WFD_TRANSITION_AT_VSYNC` value will cause the transition to occur at the next VSYNC time.

If multiple calls to **wfdBindSourceToPipeline** occur before a call to **wfdDeviceCommit** is made, only the final bind request will occur. All requests prior to the final bind request will be lost. The same property applies to mask requests, **wfdBindMaskToPipeline**.

Once a transition has started, via a call to **wfdDeviceCommit**, it can not be pre-empted as this may cause undesirable artifacts in the output.

All transitions generate asynchronous events upon completion. See section 3.6 for details.

5.7 Pipeline Configuration

For each `WFDPipeline` there are a number of attributes for status information and configuration. These attributes allow for configuration of common graphics pipeline features. Many of these attributes are read-only, thus they only provide status information. Other attributes are read-writable and provide for user control.

Configuration attribute modifications are not immediately applied to the hardware. They are cached until the user explicitly commits the changes. The user must call **wfdDeviceCommit** in order to commit the changes. All retrieved attributes will reflect the cached values, if a cached value exists for the attribute. When a pipeline is initially created, the cache of pipeline configuration modifications will be empty.

Unless otherwise noted, the read-only attributes of the pipeline configuration are static while associated pipeline handle, `WFDPipeline`, is valid.

The pixel format for a pipeline is defined by the successfully bound image or stream, thus is not otherwise configurable.

5.7.1 Pipeline Configuration Attributes

The `WFDPipelineConfigAttrib` enumeration lists the pipeline configuration attributes.

```
typedef enum
{ WFD_PIPELINE_ID                          = 0x7720,
  WFD_PIPELINE_PORTID                      = 0x7721,
  WFD_PIPELINE_LAYER                       = 0x7722,
  WFD_PIPELINE_SHAREABLE                   = 0x7723,
  WFD_PIPELINE_DIRECT_REFRESH              = 0x7724,
  WFD_PIPELINE_MAX_SOURCE_SIZE             = 0x7725,
  WFD_PIPELINE_SOURCE_RECTANGLE            = 0x7726,
  WFD_PIPELINE_FLIP                        = 0x7727,
  WFD_PIPELINE_MIRROR                      = 0x7728,
  WFD_PIPELINE_ROTATION_SUPPORT            = 0x7729,
  WFD_PIPELINE_ROTATION                    = 0x772A,
  WFD_PIPELINE_SCALE_RANGE                 = 0x772B,
  WFD_PIPELINE_SCALE_FILTER                = 0x772C,
  WFD_PIPELINE_DESTINATION_RECTANGLE = 0x772D,
  WFD_PIPELINE_TRANSPARENCY_ENABLE   = 0x772E,
  WFD_PIPELINE_GLOBAL_ALPHA                = 0x772F
} WFDPipelineConfigAttrib;
```

The storage type, default value, and read-writable state for all pipeline configuration attributes are listed in ***Table 7***.

Attribute	Storage Type	R/W	Default
WFD_PIPELINE_ID	WFDint	R	
WFD_PIPELINE_PORTID	WFDint	R	
WFD_PIPELINE_LAYER	WFDint	R	
WFD_PIPELINE_SHAREABLE	WFDboolean	R	
WFD_PIPELINE_DIRECT_REFRESH	WFDboolean	R	
WFD_PIPELINE_MAX_SOURCE_SIZE	WFDint[2]	R	
WFD_PIPELINE_SOURCE_RECTANGLE	WFDint[4]	R/W	(0,0,0,0)
WFD_PIPELINE_FLIP	WFDboolean	R/W	WFD_FALSE
WFD_PIPELINE_MIRROR	WFDboolean	R/W	WFD_FALSE
WFD_PIPELINE_ROTATION_SUPPORT	WFDRotationSupport	R	
WFD_PIPELINE_ROTATION	WFDint	R/W	0
WFD_PIPELINE_SCALE_RANGE	WFDfloat[2]	R	
WFD_PIPELINE_SCALE_FILTER	WFDScaleFilter	R/W	NONE
WFD_PIPELINE_DESTINATION_RECTANGLE	WFDint[4]	R/W	(0,0,0,0)
WFD_PIPELINE_TRANSPARENCY_ENABLE	WFDTransparency	R/W	NONE
WFD_PIPELINE_GLOBAL_ALPHA	WFDfloat	R/W	1.0

Table 7: Pipeline Configuration Attributes

5.7.1.1 IDs

The `WFD_PIPELINE_ID` attribute is the ID of the pipeline. This is the same value as retrieved from **wfdEnumeratePipelines**.

The `WFD_PIPELINE_PORTID` attribute is the ID of the port that the pipeline is bound to. If no port to pipeline binding exists, this attribute will be set to `WFD_INVALID_PORT_ID`. A pipeline can only be associated with one port, but a port can have bindings to more than one pipeline. This is the same value as retrieved from **wfdEnumeratePorts**.

Accessors: **wfdGetPipelineAttribi**

5.7.1.2 Pipeline Layering

When multiple pipelines are being used with a single port, the pipelines are layered one on top of the other. The layering, or combining, is always done such that the bottom most layer is combined with the next higher layer and then the resulting image is combined with the next higher layer, and so on.

The `WFD_PIPELINE_LAYER` attribute denotes the relative layering order of the pipeline in relation to the other pipelines that may be available or in use on the port. Numerically larger numbered layers represent pipelines above smaller numbered pipelines. Pipelines bound to the same port will have unique layer values.

It should not be assumed that the layering order values are static, as they may change when the port to pipeline binding changes. If, and only if, no port to pipeline binding exists, the attribute will be `WFD_INVALID_PIPELINE_LAYER`.

```
#define WFD_INVALID_PIPELINE_LAYER (0)
```

Accessors: **wfdGetPipelineAttribi**

5.7.1.3 Shareable Pipeline

The `WFD_PIPELINE_SHAREABLE` attribute denotes if the pipeline is usable across different ports of the `WFDDevice`. An individual pipeline can only be actively bound to one port at any point in time, but this flag denotes if the pipeline is capable of being used on another port.

Accessors: **wfdGetPipelineAttribi**

5.7.1.4 Direct Refresh

The `WFD_PIPELINE_DIRECT_REFRESH` attribute denotes if the pipeline is directly refreshing the hardware from the attached source and mask image buffers. If this attribute is `WFD_TRUE`, then the attached source and/or mask image is directly tied to the hardware for refreshing of the output at the port refresh rate. See section 8 for more details.

This attribute will only ever be active (`WFD_TRUE)` for image binding cases, `WFDEGLImage,` and may change after any image or stream binding activity, such as **wfdBindSourceToPipeline** or **wfdBindMaskToPipeline**.

Accessors: **wfdGetPipelineAttribi**

5.7.1.5 Maximum Source Image Size

The `WFD_PIPELINE_MAX_SOURCE_SIZE` attribute defines the maximum supported size of the source image for this pipeline. This attribute is in the form of a two integer array for which the first array element denotes the maximum width and the second array element denotes the maximum height.

Images larger than this size may only be bound to the pipeline if cropping is used to select a source rectangle within this size.

Accessors: **wfdGetPipelineAttribiv, wfdGetPipelineAttribfv**

5.7.1.6 Source Rectangle

The `WFD_PIPELINE_SOURCE_RECTANGLE` attribute defines the sub-rectangle of the input image to be used as the pipeline source data. This attribute is a four integer array in the form (*`offsetX`*, *`offsetY`*, *`width`*, *`height`*). These values are pixel-based and relative to the top left corner of the pipeline input image.

`offsetX` and *`offsetY`* are the pixel offset from the upper-left corner of the input image to the upper-left corner of the source rectangle. If *`offsetX`* is zero, the left edge of the input image is the left edge of the source rectangle. If *`offsetY`* is zero, the top edge of the input image is the top edge of the source rectangle.

`width` and *`height`* define the size of the source rectangle in pixels, such that (*`offsetX + width`*) and (*`offsetY + height`*) are the pixel offset from the upper-left corner of the input image to the lower-right corner of the source rectangle.

When **wfdDeviceCommit** is called, the source rectangle must be fully contained inside the input image for the configuration to be considered consistent. As a consequence, *`offsetX`* and *`offsetY`* must not be negative and must be no greater than one less than the input image width and height, respectively. Also, (*`offsetX + width`*) and (*`offsetY + height`*) must be no greater than the input image width and height, respectively.

When using fractional values for the width or height of a source rectangle that approaches the right or bottom edges of the input image, it is important for the user to verify that the resultant bottom right coordinate is in fact within the bounds of the input image. Inaccuracies in floating-point calculations can lead to non-intuitive results where the source rectangle fractionally exceeds the input 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.

If the `width` or `height` values are less than or equal to zero the pipeline will be disabled and not have any impact on the final output of the associated port.

Accessors: **wfdGetPipelineAttribiv, wfdGetPipelineAttribfv, wfdSetPipelineAttribiv, wfdSetPipelineAttribfv,**

5.7.1.7 Flip & Mirror

The `WFD_PIPELINE_FLIP` attribute denotes whether the flip feature is enabled (`WFD_TRUE`) or not (`WFD_FALSE`). Flipping consists of inverting the Crop stage output image top-to-bottom about the horizontal centerline of the image.

The `WFD_PIPELINE_MIRROR` attribute denotes whether the mirror feature is enabled (`WFD_TRUE`) or not (`WFD_FALSE`). Mirroring consists of inverting the Crop stage output image left-to-right about the vertical centerline of the cropped image.

Accessors: **wfdGetPipelineAttribi, wfdSetPipelineAttribi**

5.7.1.8 Rotation

The `WFD_PIPELINE_ROTATION_SUPPORT` attribute denotes which rotation values are supported by the pipeline. This attribute is in the form of an enumeration, `WFDRotationSupport`. If this attribute is set to "`LIMITED`", the clockwise rotation values of 0, 90, 180, and 270 degrees are supported. If this attribute is set to "`NONE`", then the only supported rotation value is 0 degrees, hence there is no rotation support. The `WFD_PIPELINE_ROTATION` attribute is used to set the desired rotation value.

Accessors (`ROTATION_SUPPORT`): **wfdGetPipelineAttribi**
Accessors (`ROTATION`): **wfdGetPipelineAttribi, wfdSetPipelineAttribi**

5.7.1.9 Scaling Range

The `WFD_PIPELINE_SCALE_RANGE` attribute denotes valid range of source image scaling supported by the pipeline. This attribute is in the form of a two floating-point value array for which the first array element denotes the minimum supported scaling factor and the second array element denotes the maximum supported scaling factor.

The scaling factor values are defined as:

`Scaling Factor = OutputSize / InputSize`

Where the *`InputSize`* is defined as either the width or height of the source rectangle and the *`OutputSize`* is defined as the corresponding width or height of the destination rectangle. Note that the scaling of the width and the height is not required to be proportional, but the limits of the scaling factor apply to both width and height.

If scaling is not supported, both the minimum and maximum values will be 1.00.

If scaling is supported, the maximum scaling factor will be 8.00 or greater and the minimum scaling factor will be 0.25 or smaller.

In some cases, for implementation specific reasons, when scaling the source rectangle to fit into the destination rectangle, the scaled source rectangle may need to be cropped to exactly fit the destination rectangle. If cropping is used, no more than 5% of the source image area will be cropped. Note that this silent scaling crop is independent of user-controlled cropping defined by the source rectangle.

Accessors: **wfdGetPipelineAttribfv**

5.7.1.10 Scaling Filter

When scaling the user can specify a filtering preference via the `WFD_PIPELINE_SCALE_FILTER` attribute. Filtering involves determining an approximate value for each image pixel using a function of the nearby pixel center values.

The `WFDScaleFilter` enumeration defines values for each type of filtering.

```
typedef enum
{ WFD_SCALE_FILTER_NONE   = 0x7760,
  WFD_SCALE_FILTER_FASTER = 0x7761,
  WFD_SCALE_FILTER_BETTER = 0x7762
} WFDScaleFilter;
```

If `WFD_PIPELINE_SCALE_FILTER` is `WFD_SCALE_FILTER_NONE`, filtering is disabled. Pixel values are determined using point sampling (also known as nearest-neighbor replication) only.

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

If `WFD_PIPELINE_SCALE_FILTER` is `WFD_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. The result of filtering a particular source point must be determined solely by the contents of the source image and the selected filtering mode[10]. 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, e.g. edge replication.

Accessors: **wfdGetPipelineAttribi, wfdSetPipelineAttribi**

5.7.1.11 Destination Rectangle

The `WFD_PIPELINE_DESTINATION_RECTANGLE` attribute defines the placement of the pipeline output image within the associated port output image. This attribute is a four integer array in the form (*`offsetX`*, *`offsetY`*, *`width`*,

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

`height`). These values are pixel-based and relative to the top left corner of the associated port output image.

`offsetX` and `offsetY` are the pixel offset from the upper-left corner of the port output image to the upper-left corner of the destination rectangle. If `offsetX` is zero, the left edge of the port output image is the left edge of the destination rectangle. If `offsetY` is zero, the top edge of the port output image is the top edge of the destination rectangle.

`width` and `height` define the size of the destination rectangle in pixels, after rotation, such that (`offsetX + width`) and (`offsetY + height`) are the pixel offset from the upper-left corner of the port output image to the lower-right corner of the destination rectangle.

When **wfdDeviceCommit** is called, the destination rectangle must be fully contained inside the port output image for the configuration to be considered consistent. As a consequence, `offsetX` and `offsetY` must not be negative and must be no greater than one less than the output image width and height, respectively. Also, (`offsetX + width`) and (`offsetY + height`) must be no greater than the output image width and height, respectively.

When using fractional values for the width or height of a destination rectangle that approaches the right or bottom edges of the port output, it is important for the user to verify that the resultant bottom right coordinate is in fact within the bounds of the port output. Inaccuracies in floating-point calculations can lead to non-intuitive results where the destination rectangle fractionally exceeds the port output 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.

If the `width` or `height` values are less than or equal to zero the pipeline will be disabled and not have any impact on the final output of the associated port.

Accessors: **wfdGetPipelineAttribiv, wfdGetPipelineAttribfv, wfdSetPipelineAttribiv, wfdSetPipelineAttribfv**

5.7.1.12 Transparency Enable

The `WFD_PIPELINE_TRANSPARENCY_ENABLE` attribute is used to define the active transparency features. The possible transparency features are defined in the `WFDTransparency` bitmask enumeration. See section 5.8 for transparency details. Attempting to enable a non-supported transparency feature combination will result in a `WFD_ERROR_ILLEGAL_ARGUMENT` failure.

Accessors: **wfdGetPipelineAttribi, wfdSetPipelineAttribi**

5.7.1.13 Global Alpha

If global alpha transparency is enabled the `WFD_PIPELINE_GLOBAL_ALPHA` attribute denotes a single opacity value that affects the blending of the pipeline data. This attribute is a single `WFDfloat` value with a range of 0 to 1.

If accessed via the **wfdSet/GetPipelineAttribi** attribute functions, the attribute is limited to the range of 0 to 255 where 0 represents fully transparent and 255 represents fully opaque. When set the passed value is conceptually divided by 255.0f to obtain an alpha value between 0 and 1. When retrieved the alpha value is clamped to the [0, 1] range, multiplied by 255 and rounded to obtain an 8-bit integer.

The source alpha and mask alpha may also affect the final transparency, see section 5.8 for details.

Accessors: **wfdGetPipelineAttribi, wfdSetPipelineAttribi, wfdGetPipelineAttribf, wfdSetPipelineAttribf**

5.7.2 Querying a Pipeline Configuration Attribute

A pipeline configuration attribute can be queried by calling:

```
WFDint wfdGetPipelineAttribi
                       ( WFDDevice              device,
                         WFDPipeline            pipeline,
                         WFDPipelineConfigAttrib attrib );

WFDfloat wfdGetPipelineAttribf
                       ( WFDDevice               device,
                         WFDPipeline             pipeline,
                         WFDPipelineConfigAttrib  attrib );

void wfdGetPipelineAttribiv
                       ( WFDDevice               device,
                         WFDPipeline             pipeline,
                         WFDPipelineConfigAttrib  attrib,
                         WFDint                  count,
                         WFDint                 *value );

void wfdGetPipelineAttribfv
                       ( WFDDevice               device,
                         WFDPipeline             pipeline,
                         WFDPipelineConfigAttrib  attrib,
                         WFDint                  count,
                         WFDfloat               *value );
```

The *device* and *pipeline* parameters denote the specific device and graphics pipeline, respectively, associated with this function call.

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 *value* must be an array of at least *count* elements.

On success, the value(s) of *attrib* for *pipeline* is returned via the function return value or the *value* parameter.

On failure, one of the errors below will be raised. Additionally, for vector based functions, *value* is not modified.

If cached attributes exist, via a call to **wfdSetPipelineAttribi** without a following call to **wfdDeviceCommit**, the cached attributes will be returned by this function.

ERRORS

`WFD_ERROR_ILLEGAL_ARGUMENT`
- if *`value`* is `NULL`
- if *`count`* does not match the attribute element count

`WFD_ERROR_BAD_HANDLE`
- if *`pipeline`* is not a valid `WFDPipeline` for *`device`*

`WFD_ERROR_BAD_ATTRIBUTE`
- if *`attrib`* is not a valid pipeline configuration attribute
- if *`attrib`* does not permit the use of the accessor

5.7.3 Setting a Pipeline Configuration Attribute

A pipeline configuration attribute can be set by calling:

```
void wfdSetPipelineAttribi
                        ( WFDDevice                device,
                          WFDPipeline              pipeline,
                          WFDPipelineConfigAttrib attrib,
                          WFDint                   value );

void wfdSetPipelineAttribf
                        ( WFDDevice                device,
                          WFDPipeline              pipeline,
                          WFDPipelineConfigAttrib attrib,
                          WFDfloat                 value );

void wfdSetPipelineAttribiv
                        ( WFDDevice                device,
                          WFDPipeline              pipeline,
                          WFDPipelineConfigAttrib attrib,
                          WFDint                   count,
                          const WFDint            *value );

void wfdSetPipelineAttribfv
                        ( WFDDevice                device,
                          WFDPipeline              pipeline,
                          WFDPipelineConfigAttrib attrib,
                          WFDint                   count,
                          const WFDfloat          *value );
```

The *`device`* and *`pipeline`* parameters denote the specific device and graphics pipeline, respectively, associated with this function call.

The *`attrib`* parameter denotes the attribute to set to *`value`*. For the vector based functions, *`count`* denotes the number of values provided and *`value`* must be an array of at least *`count`* elements.

On success, the value(s) of *`attrib`* for *`pipeline`* are set to *`value`*.

On failure, the pipeline configuration attribute is not modified and one of the errors below will be raised.

Changes to configuration attributes will not be immediately applied to the hardware. The changes are cached until **wfdDeviceCommit** is called.

ERRORS

`WFD_ERROR_ILLEGAL_ARGUMENT`
- if *`value`* is invalid or out-of-range
- if *`count`* does not match the attribute element count

`WFD_ERROR_BAD_HANDLE`
- if *`pipeline`* is not a valid `WFDPipeline` for *`device`*

`WFD_ERROR_BAD_ATTRIBUTE`
- if *`attrib`* is not a valid pipeline configuration attribute
- if *`attrib`* denotes a read-only attribute
- if *`attrib`* does not permit the use of the accessor

5.8 Transparency

For each pipeline available for a `WFDPort`, there exist a number of possible transparency features that define how the graphics image from one pipeline will be combined with the graphics image of those pipelines layered below it. These features can be used individually or in combination. The possible transparency features are defined in the `WFDTransparency` bitmask enumeration.

```
typedef enum
{ WFD_TRANSPARENCY_NONE         = 0,
  WFD_TRANSPARENCY_SOURCE_COLOR = (1 << 0),
  WFD_TRANSPARENCY_GLOBAL_ALPHA = (1 << 1),
  WFD_TRANSPARENCY_SOURCE_ALPHA = (1 << 2),
  WFD_TRANSPARENCY_MASK         = (1 << 3)
} WFDTransparency;
```

The `WFD_TRANSPARENCY_SOURCE_COLOR` feature implies the definition of fully transparent pixels in the pipeline graphics source image based on a user provided color. When this feature is enabled, all pipeline source image pixels that match the defined transparent source color will be considered transparent and thus will not be combined with the corresponding pixels of the pipelines layered below it.

The `WFD_TRANSPARENCY_GLOBAL_ALPHA` feature implies the definition of a single alpha blending value which will be used when blending each pixel defined in the pipeline with those of the pipelines layered below it.

The `WFD_TRANSPARENCY_SOURCE_ALPHA` feature implies the use of the alpha channel within the pipeline graphics image to define the blending characteristics of each pixel in the pipeline with those of the pipelines layered below it.

The `WFD_TRANSPARENCY_MASK` feature implies the use of a separate graphics image or stream attached to the pipeline to define the blending characteristics of each pixel in the pipeline with those of the pipelines layered below it.

The individual transparency features supported and the possible combination of these features are retrievable via a transparency list. This list consists of an array of integer values, such that each element in the list denotes a specific supported transparency feature or combination of features. Each list element is a bitmask containing one or more of the `WFDTransparency` enumeration values. The list can contain no elements, thus implying that the specific pipeline supports none of the transparency features.

The number of supported transparency features is static while associated pipeline handle, `WFDPipeline`, is valid.

5.8.1 Blending Functions

The effect of using transparency is defined in terms of a color blending function $c'(c'_{src}, c'_{dst}, \alpha_{src})$ where $c' = \alpha * c$ is a premultiplied color. Given a premultiplied color and alpha source tuple $(R_{src}, G_{src}, B_{src}, \alpha_{src})$ and a premultiplied color and alpha destination tuple $(R_{dst}, G_{dst}, B_{dst}, 1)$, blending replaces the destination with the blended tuple $(c'(R_{src}, R_{dst}, \alpha_{src}), c'(G_{src}, G_{dst}, \alpha_{srct}), c'(B_{src}, B_{dst}, \alpha_{src}), 1)$.

The source is converted to the destination color space prior to blending. If the source is stored in a non-premultiplied format, the alpha value is multiplied into each color value prior to applying the blending functions. If the source format does not contain an alpha channel, an alpha value of 1 is used in place of α_{src}. The destination will always have an alpha value of 1.

Source transparent color values must contain alpha channel values of 1 or the color match may fail.

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} represents the premultiplied destination color
- c_{trans} represents the transparent source color value
- a_{gbl} represents the global alpha value
- a_{mask} represents the alpha value from the image mask

The blending functions for each transparency mode are as follows.

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

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

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

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

`WFD_TRANSPARENCY_GLOBAL_ALPHA | SOURCE_ALPHA`
$c'(c'_{src}, c'_{dst}, a_{src}) = c'_{src} * a_{gbl} + c'_{dst} * (1 - a_{src} * a_{gbl})$

`WFD_TRANSPARENCY_GLOBAL_ALPHA | MASK`
$c'(c'_{src}, c'_{dst}, a_{src}) = c'_{src} * a_{mask} * a_{gbl} + c'_{dst} * (1 - a_{mask} * a_{gbl})$

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

`WFD_TRANSPARENCY_GLOBAL_ALPHA | SOURCE_ALPHA | MASK`
$c'(c'_{src}, c'_{dst}, a_{src}) = c'_{src} * a_{gbl} * a_{mask} + c'_{dst} * (1 - a_{src} * a_{gbl} * a_{mask})$

`WFD_TRANSPARENCY_SOURCE_COLOR` *`(in combination with any other case)`*

```
If ( $c'_{src}$ equals $c_{trans}$ )
  $c'(c'_{src}, c'_{dst}, a_{src}) = c'_{dst}$
Else
  <the other transparency function>
Endif
```

5.8.2 Querying Transparency Combinations

The transparency features supported by a graphics pipeline can be retrieved by calling:

```
WFDint wfdGetPipelineTransparency
                                ( WFDDevice    device,
                                  WFDPipeline  pipeline,
                                  WFDbitfield *trans,
                                  WFDint       transCount );
```

The *device* and *pipeline* parameters denote the specific device and graphics pipeline, respectively, associated with this function call.

The *trans* and *transCount* parameters define the caller provided array to return the transparency features into.

On success, the number of *trans* elements that were populated with transparency feature elements is returned via the function return value. Thus elements 0 through (the returned value) – 1 of *trans* will contain valid elements. No more than *transCount* elements will be returned, even if more are available. However, if **wfdGetPipelineTransparency** is called with *trans* = NULL then no elements are returned, but the total number of available transparency feature elements is returned via the function return value.

On failure, *trans* is not modified and one of the errors below will be raised.

ERRORS

WFD_ERROR_ILLEGAL_ARGUMENT
- if *transCount* is invalid, zero or negative

WFD_ERROR_BAD_HANDLE
- if *pipeline* is not a valid WFDPipeline for *device*

5.8.3 Transparent Source Color Definition

The definition of a transparent source color value must be done in a precise manner or the transparency effect may fail. An understanding of the other graphics engines or components involved, which is outside the scope of this document, may be required in order to accurately and predictably determine the proper transparent source color value. Given this, the scope of the source transparent color definition is limited to integer linear RGB color space values[11]. The WFDTSColorFormat enumeration contains the initially supported list.

[11] The definition is made such that expansion to other color spaces or bit patterns should be possible and relatively easy.

```
typedef enum
{ WFD_TSC_FORMAT_UINT8_RGB_8_8_8_LINEAR = 0x7790,
  WFD_TSC_FORMAT_UINT8_RGB_5_6_5_LINEAR = 0x7791
} WFDTSColorFormat;
```

The suggested form for extensions to this enumeration is:

`WFD_TSC_FORMAT_`<*data type*>_<*color space*>_<*bits per element*>_<*misc*>

Where: "*data type*" defines the data type for each element; "*color space*" defines the color space for each element; "*bits per element*" defines the, underscore separated and LSB justified, number of bits per color space elements; and "*misc*" for any other miscellaneous information necessary.

For example, `WFD_TSC_FORMAT_UINT8_RGB_5_6_5_LINEAR` implies: an array of 8-bit integers; one array element each for components "R", "G", and "B"; the "R" component is the lower 5 bits of the first array element, the "G" component is the lower 6 bits of the second array element, etc.; and all components are linear values.

The source transparent color value can be specified by calling:

```
void wfdSetPipelineTSColor( WFDDevice        device,
                            WFDPipeline      pipeline,
                            WFDTSColorFormat colorFormat,
                            WFDint           count,
                            const void      *color );
```

The *`device`* and *`pipeline`* parameters denote the specific device and pipeline, respectively, associated with this function call.

The *`colorFormat`* parameter denotes the format of the source transparent color data array defined by the array pointer, *`color`*, and the element count, *`count`*.

On success, the transparent source color value for *`pipeline`* will be set according to *`colorFormat`*, *`color`*, and *`count`*.

On failure, *`pipeline`* will not be modified and one of the below errors will be generated.

When a pipeline is created the transparent source color value will be defaulted to all zeros. If source color transparency is enabled prior to setting the color value, the default value will be used.

ERRORS

`WFD_ERROR_ILLEGAL_ARGUMENT`
- if *`colorFormat`* is not a valid `WFDTSColorFormat`
- if *`color`* is `NULL`
- if *`count`* does not match the size of *`colorFormat`*

`WFD_ERROR_NOT_SUPPORTED`
- if source color transparency is not supported by the pipeline

`WFD_ERROR_BAD_HANDLE`
- if *`pipeline`* is not a valid `WFDPipeline` for *`device`*

5.9 Querying the Layer number for any Port

Only when a pipeline is bound to a port can the pipelines relative layering order value, for that specific port, be read via the pipeline configuration attributes, see section 5.7.1.2. There are times, especially during a cached setup, that this layering order value for a specific, not yet bound, port is important to know.

The layering order value for a pipeline, in relation to a specific port, can be retrieved by calling:

```
WFDint wfdGetPipelineLayerOrder( WFDDevice   device,
                                 WFDPort     port,
                                 WFDPipeline pipeline );
```

The *`device`*, *`port`*, and *`pipeline`* parameters denote the specific device, port, and pipeline, respectively, associated with this function call.

On success, the layering order value for *`pipeline`* with respect to binding with `port` is returned. This returned value is the same value the `WFD_PIPELINE_LAYER` attribute would have if *`port`* and *`pipeline`* were bound.

On failure, `WFD_INVALID_PIPELINE_LAYER` is returned and one of the below errors are generated.

ERRORS

`WFD_ERROR_ILLEGAL_ARGUMENT`
- if *`pipeline`* is not bindable to *`port`*

`WFD_ERROR_BAD_HANDLE`
- if *`port`* is not a valid `WFDPort` for *`device`*
- if *`pipeline`* is not a valid `WFDPipeline` for *`device`*

6 Extending OpenWF Display

OpenWF Display 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.

6.1 Extension Naming Conventions

An OpenWF Display API extension is named by a string of the form `WFD_`*`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 `#defines`) defined by an extension carry the suffix *`_type`*. Functions and datatypes carry the suffix *`type`* without a separating underscore.

The `<WF/wfdext.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 `WFD_`*`type_name`* and a value of 1 for each supported extension.

6.2 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 `WFD_`*`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 Display API specification
- New errors generated by functions affected by the extension
- New state defined by the extension
- Authorship information and revision history

6.3 Using Extensions

Extensions may be detected statically, by means of preprocessor symbols, or dynamically, by means of the **wfdGetStrings** or **wfdIsExtensionSupported** functions. Using static detection of extensions at compile time is generally not sufficient to insure an implementation supports a particular extension. When static detection is used, a runtime dynamic check for the extension should also be made to insure 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.

6.3.1 Accessing Extensions Statically

The extensions defined by a given platform are defined in the `<WF/wfdext.h>` header file, or in header files automatically included by `<WF/wfdext.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 WFD_EXT_my_extension
    wfdMyExtensionFuncEXT(...);
#endif
```

6.3.2 Accessing Extensions Dynamically

The OpenWF Display API 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 `WFDStringID` enumeration defines values for strings that the user can query.

```
typedef enum
{ WFD_VENDOR     = 0x7500,
  WFD_RENDERER   = 0x7501,
  WFD_VERSION    = 0x7502,
  WFD_EXTENSIONS = 0x7503
} WFDStringID;
```

The **wfdGetStrings** function returns information about the OpenWF Display implementation, including extension information.

```
WFDint wfdGetStrings( WFDDevice     device,
                      WFDStringID   name,
                      const char  **strings,
                      WFDint        stringsCount );
```

The user provides a buffer to receive a list of pointers to strings specific to *`device`*. *`strings`* is the address of the buffer. *`stringsCount`* is the number of string pointers that can fit into the buffer. If *`device`* 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.

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.

The combination of `WFD_VENDOR` and `WFD_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 `WFD_VENDOR`, a single string of the name of company responsible for this OpenWF Display implementation is returned.

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

If *`name`* is `WFD_EXTENSIONS`, a list of strings denoting the supported extensions to OpenWF Display is returned.

ERRORS

`WFD_ERROR_ILLEGAL_ARGUMENT`
- if *`stringCount`* is negative
- if *`name`* is not a valid string ID

The **wfdIsExtensionSupported** function provides an alternate means of testing for support of an extension.

```
WFDboolean wfdIsExtensionSupported( WFDDevice   device,
                                    const char *string );
```

`WFD_TRUE` will be returned if the extension denoted by *`string`* is supported by *`device`*. Otherwise `WFD_FALSE` will be returned.

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

6.4 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 Display API extension may be deemed a shared extension if two or more vendors agree in good faith to ship an extension, or the Khronos OpenWF 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 Display API specification.

7 Pre-Configuration

In some cases the display hardware may already be in use by the implementation prior to the creation of the corresponding OpenWF Display device, port, or pipeline(s). This is particularly true for display hardware which is used for device power-on sign-of-life indications. In these cases, the implementation may optionally expose this hardware pre-configuration to the user in the following manner:

1. The port mode will already be set when the port is created.

   The user will be able to determine this via a call to **wfdGetCurrentPortMode** prior to setting the port mode. If the port mode has been pre-configured, then the pre-configured port mode will be returned.

2. The port configuration will already be set when the port is created.

   Most notably, the port power mode will not be in the default "OFF" state. Other port configuration attributes may also be in non-default states.

3. One or more of the pipelines associated with the pre-configured port will already be bound to the port and in use indirectly.

   This user will be able to determine this via the pipeline "Port ID" attribute, see section 5.7.1.1.

8 Image Swapping & Updating

The cases of the non-stream base swapping or updating of pipeline image data can be somewhat hardware and/or content dependent. For example, in some cases the hardware may directly and continually refresh from the bound image which would force a multi-buffered scenario. In other cases, the content may not completely change within a newly bound image, for example with UI content where only a small portion of the image is updated, like the displayed time on a status bar. These scenarios are described in detail in the below sub-sections.

8.1 Direct Hardware Refreshing

In the case where an image, not a stream, is bound to a pipeline, it is possible that the implementation may indicate that the hardware is directly refreshing from the bound image. The implementation will indicate this case via the `WFD_PIPELINE_DIRECT_REFRESH` pipeline attribute. Having a bound image directly tied to the hardware refresh implies that any changes made to that image buffer will be immediately applied to the pipeline output. This can cause a number of undesirable side effects on the pipeline output, such as tearing or other visual artifacts.

In order to avoid these side effects, users should operate using multiple images for the affected pipeline. The user should ensure that the currently bound image is not modified. Note that the state of the `WFD_PIPELINE_DIRECT_REFRESH` pipeline attribute may change with each call to **wfdBindSourceToPipeline**.

8.2 Limited Update Region

For use cases in which the changes in the input image for a pipeline do not affect a significant portion of the image, the user may consider providing the implementation with a hint denoting what portion of the image has changed. A hint such as this may allow the implementation to perform a more efficient update of the pipeline input image.

The hint as to what portion of the image has changed is provided via the optional "region" parameter of the image binding function, **wfdBindSourceToPipeline**. This "region" parameter specifies a sub-rectangle of the passed in image which has changed from the currently bound image. This sub-rectangle must be smaller than and completely contained within the input image or a `WFD_ERROR_ILLEGAL_ARGUMENT` will be generated.

9 Appendix A: Header Files

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

wfd.h

```
/*********************************************************************
 *                                                                   *
 * Sample implementation of wfd.h, version 1.0                       *
 *                                                                   *
 * Copyright © 2007-2008 The Khronos Group                           *
 *                                                                   *
 *********************************************************************/

#ifndef _WFD_H_
#define _WFD_H_

#include <WF/wfdplatform.h>

#ifdef __cplusplus
extern "C" {
#endif

#define OPENWFD_VERSION_1_0         (1)

#define WFD_NONE                    (0)

#define WFD_INVALID_PORT_ID         (0)
#define WFD_INVALID_PIPELINE_ID     (0)
#define WFD_INVALID_PIPELINE_LAYER  (0)

#define WFD_DEFAULT_DEVICE_ID       (0)

#define WFD_MAX_INT   ((WFDint)16777216)
#define WFD_MAX_FLOAT ((WFDfloat)16777216)

#define WFD_INVALID_HANDLE ((WFDHandle)0)

typedef WFDHandle WFDDevice;
typedef WFDHandle WFDEvent;
typedef WFDHandle WFDPort;
typedef WFDHandle WFDPortMode;
typedef WFDHandle WFDPipeline;
typedef WFDHandle WFDSource;
typedef WFDHandle WFDMask;

typedef enum
{ WFD_VENDOR                                    = 0x7500,
  WFD_RENDERER                                  = 0x7501,
  WFD_VERSION                                   = 0x7502,
  WFD_EXTENSIONS                                = 0x7503,
  WFD_STRING_ID_FORCE_32BIT                     = 0x7FFFFFFF
```

```
} WFDStringID;

typedef enum
{ WFD_ERROR_NONE                              = 0,
  WFD_ERROR_OUT_OF_MEMORY                     = 0x7510,
  WFD_ERROR_ILLEGAL_ARGUMENT                  = 0x7511,
  WFD_ERROR_NOT_SUPPORTED                     = 0x7512,
  WFD_ERROR_BAD_ATTRIBUTE                     = 0x7513,
  WFD_ERROR_IN_USE                            = 0x7514,
  WFD_ERROR_BUSY                              = 0x7515,
  WFD_ERROR_BAD_DEVICE                        = 0x7516,
  WFD_ERROR_BAD_HANDLE                        = 0x7517,
  WFD_ERROR_INCONSISTENCY                     = 0x7518,
  WFD_ERROR_FORCE_32BIT                       = 0x7FFFFFFF
} WFDErrorCode;

typedef enum
{ WFD_DEVICE_FILTER_PORT_ID                   = 0x7530,
  WFD_DEVICE_FILTER_FORCE_32BIT               = 0x7FFFFFFF
} WFDDeviceFilter;

typedef enum
{ WFD_COMMIT_ENTIRE_DEVICE                    = 0x7550,
  WFD_COMMIT_ENTIRE_PORT                      = 0x7551,
  WFD_COMMIT_PIPELINE                         = 0x7552,
  WFD_COMMIT_FORCE_32BIT                      = 0x7FFFFFFF
} WFDCommitType;

typedef enum
{ WFD_DEVICE_ID                               = 0x7560,
  WFD_DEVICE_ATTRIB_FORCE_32BIT               = 0x7FFFFFFF
} WFDDeviceAttrib;

typedef enum
{ WFD_EVENT_INVALID                           = 0x7580,
  WFD_EVENT_NONE                              = 0x7581,
  WFD_EVENT_DESTROYED                         = 0x7582,
  WFD_EVENT_PORT_ATTACH_DETACH                = 0x7583,
  WFD_EVENT_PORT_PROTECTION_FAILURE           = 0x7584,
  WFD_EVENT_PIPELINE_BIND_SOURCE_COMPLETE     = 0x7585,
  WFD_EVENT_PIPELINE_BIND_MASK_COMPLETE       = 0x7586,
  WFD_EVENT_FORCE_32BIT                       = 0x7FFFFFFF
} WFDEventType;

typedef enum
{ /* Configuration Attributes */
  WFD_EVENT_PIPELINE_BIND_QUEUE_SIZE          = 0x75C0,

  /* Generic Event Attributes */
  WFD_EVENT_TYPE                              = 0x75C1,

  /* Port Attach Event Attributes */
  WFD_EVENT_PORT_ATTACH_PORT_ID               = 0x75C2,
  WFD_EVENT_PORT_ATTACH_STATE                 = 0x75C3,

  /* Port Protection Event Attributes */
  WFD_EVENT_PORT_PROTECTION_PORT_ID           = 0x75C4,
  /* Pipeline Bind Complete Event Attributes */
  WFD_EVENT_PIPELINE_BIND_PIPELINE_ID       = 0x75C5,
  WFD_EVENT_PIPELINE_BIND_SOURCE            = 0x75C6,
  WFD_EVENT_PIPELINE_BIND_MASK              = 0x75C7,
  WFD_EVENT_PIPELINE_BIND_QUEUE_OVERFLOW    = 0x75C8,

  WFD_EVENT_ATTRIB_FORCE_32BIT              = 0x7FFFFFFF
} WFDEventAttrib;

typedef enum
{ WFD_PORT_MODE_WIDTH                       = 0x7600,
  WFD_PORT_MODE_HEIGHT                      = 0x7601,
  WFD_PORT_MODE_REFRESH_RATE                = 0x7602,
  WFD_PORT_MODE_FLIP_MIRROR_SUPPORT         = 0x7603,
  WFD_PORT_MODE_ROTATION_SUPPORT            = 0x7604,
  WFD_PORT_MODE_INTERLACED                  = 0x7605,
  WFD_PORT_MODE_ATTRIB_FORCE_32BIT          = 0x7FFFFFFF
} WFDPortModeAttrib;

typedef enum
{ WFD_PORT_ID                               = 0x7620,
  WFD_PORT_TYPE                             = 0x7621,
  WFD_PORT_DETACHABLE                       = 0x7622,
  WFD_PORT_ATTACHED                         = 0x7623,
  WFD_PORT_NATIVE_RESOLUTION                = 0x7624,
  WFD_PORT_PHYSICAL_SIZE                    = 0x7625,
  WFD_PORT_FILL_PORT_AREA                   = 0x7626,
  WFD_PORT_BACKGROUND_COLOR                 = 0x7627,
  WFD_PORT_FLIP                             = 0x7628,
  WFD_PORT_MIRROR                           = 0x7629,
  WFD_PORT_ROTATION                         = 0x762A,
  WFD_PORT_POWER_MODE                       = 0x762B,
  WFD_PORT_GAMMA_RANGE                      = 0x762C,
  WFD_PORT_GAMMA                            = 0x762D,
  WFD_PORT_PARTIAL_REFRESH_SUPPORT          = 0x762E,
  WFD_PORT_PARTIAL_REFRESH_MAXIMUM          = 0x762F,
  WFD_PORT_PARTIAL_REFRESH_ENABLE           = 0x7630,
  WFD_PORT_PARTIAL_REFRESH_RECTANGLE        = 0x7631,
  WFD_PORT_PIPELINE_ID_COUNT                = 0x7632,
  WFD_PORT_BINDABLE_PIPELINE_IDS            = 0x7633,
  WFD_PORT_PROTECTION_ENABLE                = 0x7634,
  WFD_PORT_ATTRIB_FORCE_32BIT               = 0x7FFFFFFF
} WFDPortConfigAttrib;

typedef enum
{ WFD_PORT_TYPE_INTERNAL                    = 0x7660,
  WFD_PORT_TYPE_COMPOSITE                   = 0x7661,
  WFD_PORT_TYPE_SVIDEO                      = 0x7662,
  WFD_PORT_TYPE_COMPONENT_YPbPr             = 0x7663,
  WFD_PORT_TYPE_COMPONENT_RGB               = 0x7664,
  WFD_PORT_TYPE_COMPONENT_RGBHV             = 0x7665,
  WFD_PORT_TYPE_DVI                         = 0x7666,
  WFD_PORT_TYPE_HDMI                        = 0x7667,
  WFD_PORT_TYPE_DISPLAYPORT                 = 0x7668,
  WFD_PORT_TYPE_OTHER                       = 0x7669,
  WFD_PORT_TYPE_FORCE_32BIT                 = 0x7FFFFFFF
```

```
} WFDPortType;

typedef enum
{ WFD_POWER_MODE_OFF                            = 0x7680,
  WFD_POWER_MODE_SUSPEND                        = 0x7681,
  WFD_POWER_MODE_LIMITED_USE                    = 0x7682,
  WFD_POWER_MODE_ON                             = 0x7683,
  WFD_POWER_MODE_FORCE_32BIT                    = 0x7FFFFFFF
} WFDPowerMode;

typedef enum
{ WFD_PARTIAL_REFRESH_NONE                      = 0x7690,
  WFD_PARTIAL_REFRESH_VERTICAL                  = 0x7691,
  WFD_PARTIAL_REFRESH_HORIZONTAL                = 0x7692,
  WFD_PARTIAL_REFRESH_BOTH                      = 0x7693,
  WFD_PARTIAL_REFRESH_FORCE_32BIT               = 0x7FFFFFFF
} WFDPartialRefresh;

typedef enum
{ WFD_DISPLAY_DATA_FORMAT_NONE                  = 0x76A0,
  WFD_DISPLAY_DATA_FORMAT_EDID_V1               = 0x76A1,
  WFD_DISPLAY_DATA_FORMAT_EDID_V2               = 0x76A2,
  WFD_DISPLAY_DATA_FORMAT_DISPLAYID             = 0x76A3,
  WFD_DISPLAY_DATA_FORMAT_FORCE_32BIT           = 0x7FFFFFFF
} WFDDisplayDataFormat;

typedef enum
{ WFD_ROTATION_SUPPORT_NONE                     = 0x76D0,
  WFD_ROTATION_SUPPORT_LIMITED                  = 0x76D1,
  WFD_ROTATION_SUPPORT_FORMAT_FORCE_32BIT       = 0x7FFFFFFF
} WFDRotationSupport;

typedef enum
{ WFD_PIPELINE_ID                               = 0x7720,
  WFD_PIPELINE_PORTID                           = 0x7721,
  WFD_PIPELINE_LAYER                            = 0x7722,
  WFD_PIPELINE_SHAREABLE                        = 0x7723,
  WFD_PIPELINE_DIRECT_REFRESH                   = 0x7724,
  WFD_PIPELINE_MAX_SOURCE_SIZE                  = 0x7725,
  WFD_PIPELINE_SOURCE_RECTANGLE                 = 0x7726,
  WFD_PIPELINE_FLIP                             = 0x7727,
  WFD_PIPELINE_MIRROR                           = 0x7728,
  WFD_PIPELINE_ROTATION_SUPPORT                 = 0x7729,
  WFD_PIPELINE_ROTATION                         = 0x772A,
  WFD_PIPELINE_SCALE_RANGE                      = 0x772B,
  WFD_PIPELINE_SCALE_FILTER                     = 0x772C,
  WFD_PIPELINE_DESTINATION_RECTANGLE            = 0x772D,
  WFD_PIPELINE_TRANSPARENCY_ENABLE              = 0x772E,
  WFD_PIPELINE_GLOBAL_ALPHA                     = 0x772F,
  WFD_PIPELINE_ATTRIB_FORCE_32BIT               = 0x7FFFFFFF
} WFDPipelineConfigAttrib;

typedef enum
{ WFD_SCALE_FILTER_NONE                         = 0x7760,
  WFD_SCALE_FILTER_FASTER                       = 0x7761,
  WFD_SCALE_FILTER_BETTER                       = 0x7762,
  WFD_SCALE_FILTER_FORCE_32BIT                  = 0x7FFFFFFF
```

```
} WFDScaleFilter;

typedef enum
{ WFD_TRANSPARENCY_NONE                         = 0,
  WFD_TRANSPARENCY_SOURCE_COLOR                 = (1 << 0),
  WFD_TRANSPARENCY_GLOBAL_ALPHA                 = (1 << 1),
  WFD_TRANSPARENCY_SOURCE_ALPHA                 = (1 << 2),
  WFD_TRANSPARENCY_MASK                         = (1 << 3),
  WFD_TRANSPARENCY_FORCE_32BIT                  = 0x7FFFFFFF
} WFDTransparency;

typedef enum
{ WFD_TSC_FORMAT_UINT8_RGB_8_8_8_LINEAR     = 0x7790,
  WFD_TSC_FORMAT_UINT8_RGB_5_6_5_LINEAR     = 0x7791,
  WFD_TSC_FORMAT_FORCE_32BIT                = 0x7FFFFFFF
} WFDTSColorFormat;

typedef enum
{ WFD_TRANSITION_INVALID                        = 0x77E0,
  WFD_TRANSITION_IMMEDIATE                      = 0x77E1,
  WFD_TRANSITION_AT_VSYNC                       = 0x77E2,
  WFD_TRANSITION_FORCE_32BIT                    = 0x7FFFFFFF
} WFDTransition;

typedef struct
{ WFDint offsetX;
  WFDint offsetY;
  WFDint width;
  WFDint height;
} WFDRect;

/* Function Prototypes */

/* Implementation Information */

WFD_API_CALL WFDint WFD_APIENTRY
    wfdGetStrings(WFDDevice device,
                  WFDStringID name,
                  const char **strings,
                  WFDint stringsCount) WFD_APIEXIT;

WFD_API_CALL WFDboolean WFD_APIENTRY
    wfdIsExtensionSupported(WFDDevice device,
                            const char *string) WFD_APIEXIT;

/* Error */

WFD_API_CALL WFDErrorCode WFD_APIENTRY
    wfdGetError(WFDDevice device) WFD_APIEXIT;

/* Device */

WFD_API_CALL WFDint WFD_APIENTRY
    wfdEnumerateDevices(WFDint *deviceIds,
                        WFDint deviceIdsCount,
                        const WFDint *filterList) WFD_APIEXIT;
```

```
WFD_API_CALL WFDDevice WFD_APIENTRY
    wfdCreateDevice(WFDint deviceId,
                    const WFDint *attribList) WFD_APIEXIT;

WFD_API_CALL WFDErrorCode WFD_APIENTRY
    wfdDestroyDevice(WFDDevice device) WFD_APIEXIT;

WFD_API_CALL void WFD_APIENTRY
    wfdDeviceCommit(WFDDevice device,
                    WFDCommitType type,
                    WFDHandle handle) WFD_APIEXIT;

WFD_API_CALL WFDint WFD_APIENTRY
    wfdGetDeviceAttribi(WFDDevice device,
                        WFDDeviceAttrib attrib) WFD_APIEXIT;

WFD_API_CALL void WFD_APIENTRY
    wfdSetDeviceAttribi(WFDDevice device,
                        WFDDeviceAttrib attrib,
                        WFDint value) WFD_APIEXIT;

WFD_API_CALL WFDEvent WFD_APIENTRY
    wfdCreateEvent(WFDDevice device,
                   const WFDint *attribList) WFD_APIEXIT;

WFD_API_CALL void WFD_APIENTRY
    wfdDestroyEvent(WFDDevice device,
                    WFDEvent event) WFD_APIEXIT;

WFD_API_CALL WFDint WFD_APIENTRY
    wfdGetEventAttribi(WFDDevice device,
                       WFDEvent event,
                       WFDEventAttrib attrib) WFD_APIEXIT;

WFD_API_CALL void WFD_APIENTRY
    wfdDeviceEventAsync(WFDDevice device,
                        WFDEvent event,
                        WFDEGLDisplay display,
                        WFDEGLSync sync) WFD_APIEXIT;

WFD_API_CALL WFDEventType WFD_APIENTRY
    wfdDeviceEventWait(WFDDevice device,
                       WFDEvent event,
                       WFDtime timeout) WFD_APIEXIT;

WFD_API_CALL void WFD_APIENTRY
    wfdDeviceEventFilter(WFDDevice device,
                         WFDEvent event,
                         const WFDEventType *filter) WFD_APIEXIT;

/* Port */

WFD_API_CALL WFDint WFD_APIENTRY
    wfdEnumeratePorts(WFDDevice device,
                      WFDint *portIds,
                      WFDint portIdsCount,
                      const WFDint *filterList) WFD_APIEXIT;
```

```
WFD_API_CALL WFDPort WFD_APIENTRY
    wfdCreatePort(WFDDevice device,
                  WFDint portId,
                  const WFDint *attribList) WFD_APIEXIT;

WFD_API_CALL void WFD_APIENTRY
    wfdDestroyPort(WFDDevice device, WFDPort port) WFD_APIEXIT;

WFD_API_CALL WFDint WFD_APIENTRY
    wfdGetPortModes(WFDDevice device,
                    WFDPort port,
                    WFDPortMode *modes,
                    WFDint modesCount) WFD_APIEXIT;

WFD_API_CALL WFDint WFD_APIENTRY
    wfdGetPortModeAttribi(WFDDevice device,
                          WFDPort port,
                          WFDPortMode mode,
                          WFDPortModeAttrib attrib) WFD_APIEXIT;

WFD_API_CALL WFDfloat WFD_APIENTRY
    wfdGetPortModeAttribf(WFDDevice device,
                          WFDPort port,
                          WFDPortMode mode,
                          WFDPortModeAttrib attrib) WFD_APIEXIT;

WFD_API_CALL void WFD_APIENTRY
    wfdSetPortMode(WFDDevice device,
                   WFDPort port,
                   WFDPortMode mode) WFD_APIEXIT;

WFD_API_CALL WFDPortMode WFD_APIENTRY
    wfdGetCurrentPortMode(WFDDevice device, WFDPort port) WFD_APIEXIT;

WFD_API_CALL WFDint WFD_APIENTRY
    wfdGetPortAttribi(WFDDevice device,
                      WFDPort port,
                      WFDPortConfigAttrib attrib) WFD_APIEXIT;

WFD_API_CALL WFDfloat WFD_APIENTRY
    wfdGetPortAttribf(WFDDevice device,
                      WFDPort port,
                      WFDPortConfigAttrib attrib) WFD_APIEXIT;

WFD_API_CALL void WFD_APIENTRY
    wfdGetPortAttribiv(WFDDevice device,
                       WFDPort port,
                       WFDPortConfigAttrib attrib,
                       WFDint count,
                       WFDint *value) WFD_APIEXIT;

WFD_API_CALL void WFD_APIENTRY
    wfdGetPortAttribfv(WFDDevice device,
                       WFDPort port,
                       WFDPortConfigAttrib attrib,
                       WFDint count,
                       WFDfloat *value) WFD_APIEXIT;

WFD_API_CALL void WFD_APIENTRY
    wfdSetPortAttribi(WFDDevice device,
                      WFDPort port,
                      WFDPortConfigAttrib attrib,
                      WFDint value) WFD_APIEXIT;

WFD_API_CALL void WFD_APIENTRY
    wfdSetPortAttribf(WFDDevice device,
                      WFDPort port,
                      WFDPortConfigAttrib attrib,
                      WFDfloat value) WFD_APIEXIT;

WFD_API_CALL void WFD_APIENTRY
    wfdSetPortAttribiv(WFDDevice device,
                       WFDPort port,
                       WFDPortConfigAttrib attrib,
                       WFDint count,
                       const WFDint *value) WFD_APIEXIT;

WFD_API_CALL void WFD_APIENTRY
    wfdSetPortAttribfv(WFDDevice device,
                       WFDPort port,
                       WFDPortConfigAttrib attrib,
                       WFDint count,
                       const WFDfloat *value) WFD_APIEXIT;

WFD_API_CALL void WFD_APIENTRY
    wfdBindPipelineToPort(WFDDevice device,
                          WFDPort port,
                          WFDPipeline pipeline) WFD_APIEXIT;

WFD_API_CALL WFDint WFD_APIENTRY
    wfdGetDisplayDataFormats(WFDDevice device,
                             WFDPort port,
                             WFDDisplayDataFormat *format,
                             WFDint formatCount) WFD_APIEXIT;

WFD_API_CALL WFDint WFD_APIENTRY
    wfdGetDisplayData(WFDDevice device,
                      WFDPort port,
                      WFDDisplayDataFormat format,
                      WFDuint8 *data,
                      WFDint dataCount) WFD_APIEXIT;

/* Pipeline */

WFD_API_CALL WFDint WFD_APIENTRY
    wfdEnumeratePipelines(WFDDevice device,
                          WFDint *pipelineIds,
                          WFDint pipelineIdsCount,
                          const WFDint *filterList) WFD_APIEXIT;

WFD_API_CALL WFDPipeline WFD_APIENTRY
    wfdCreatePipeline(WFDDevice device,
                      WFDint pipelineId,
                      const WFDint *attribList) WFD_APIEXIT;

WFD_API_CALL void WFD_APIENTRY
    wfdDestroyPipeline(WFDDevice device,
                       WFDPipeline pipeline) WFD_APIEXIT;

WFD_API_CALL WFDSource WFD_APIENTRY
    wfdCreateSourceFromImage(WFDDevice device,
                             WFDPipeline pipeline,
                             WFDEGLImage image,
                             const WFDint *attribList) WFD_APIEXIT;

WFD_API_CALL WFDSource WFD_APIENTRY
    wfdCreateSourceFromStream(WFDDevice device,
                              WFDPipeline pipeline,
                              WFDNativeStreamType stream,
                              const WFDint *attribList) WFD_APIEXIT;

WFD_API_CALL void WFD_APIENTRY
    wfdDestroySource(WFDDevice device,
                     WFDSource source) WFD_APIEXIT;

WFD_API_CALL WFDMask WFD_APIENTRY
    wfdCreateMaskFromImage(WFDDevice device,
                           WFDPipeline pipeline,
                           WFDEGLImage image,
                           const WFDint *attribList) WFD_APIEXIT;

WFD_API_CALL WFDMask WFD_APIENTRY
    wfdCreateMaskFromStream(WFDDevice device,
                            WFDPipeline pipeline,
                            WFDNativeStreamType stream,
                            const WFDint *attribList) WFD_APIEXIT;

WFD_API_CALL void WFD_APIENTRY
    wfdDestroyMask(WFDDevice device,
                   WFDMask mask) WFD_APIEXIT;

WFD_API_CALL void WFD_APIENTRY
    wfdBindSourceToPipeline(WFDDevice device,
                            WFDPipeline pipeline,
                            WFDSource source,
                            WFDTransition transition,
                            const WFDRect *region) WFD_APIEXIT;

WFD_API_CALL void WFD_APIENTRY
    wfdBindMaskToPipeline(WFDDevice device,
                          WFDPipeline pipeline,
                          WFDMask mask,
                          WFDTransition transition) WFD_APIEXIT;

WFD_API_CALL WFDint WFD_APIENTRY
    wfdGetPipelineAttribi(WFDDevice device,
                          WFDPipeline pipeline,
                          WFDPipelineConfigAttrib attrib) WFD_APIEXIT;

WFD_API_CALL WFDfloat WFD_APIENTRY
```

```
    wfdGetPipelineAttribf(WFDDevice device,
                          WFDPipeline pipeline,
                          WFDPipelineConfigAttrib attrib) WFD_APIEXIT;

WFD_API_CALL void WFD_APIENTRY
    wfdGetPipelineAttribiv(WFDDevice device,
                           WFDPipeline pipeline,
                           WFDPipelineConfigAttrib attrib,
                           WFDint count,
                           WFDint *value) WFD_APIEXIT;

WFD_API_CALL void WFD_APIENTRY
    wfdGetPipelineAttribfv(WFDDevice device,
                           WFDPipeline pipeline,
                           WFDPipelineConfigAttrib attrib,
                           WFDint count,
                           WFDfloat *value) WFD_APIEXIT;

WFD_API_CALL void WFD_APIENTRY
    wfdSetPipelineAttribi(WFDDevice device,
                          WFDPipeline pipeline,
                          WFDPipelineConfigAttrib attrib,
                          WFDint value) WFD_APIEXIT;

WFD_API_CALL void WFD_APIENTRY
    wfdSetPipelineAttribf(WFDDevice device,
                          WFDPipeline pipeline,
                          WFDPipelineConfigAttrib attrib,
                          WFDfloat value) WFD_APIEXIT;

WFD_API_CALL void WFD_APIENTRY
    wfdSetPipelineAttribiv(WFDDevice device,
                           WFDPipeline pipeline,
                           WFDPipelineConfigAttrib attrib,
                           WFDint count,
                           const WFDint *value) WFD_APIEXIT;

WFD_API_CALL void WFD_APIENTRY
    wfdSetPipelineAttribfv(WFDDevice device,
                           WFDPipeline pipeline,
                           WFDPipelineConfigAttrib attrib,
                           WFDint count,
                           const WFDfloat *value) WFD_APIEXIT;

WFD_API_CALL WFDint WFD_APIENTRY
    wfdGetPipelineTransparency(WFDDevice device,
                               WFDPipeline pipeline,
                               WFDbitfield *trans,
                               WFDint transCount) WFD_APIEXIT;

WFD_API_CALL void WFD_APIENTRY
    wfdSetPipelineTSColor(WFDDevice device,
                          WFDPipeline pipeline,
                          WFDTSColorFormat colorFormat,
                          WFDint count,
                          const void *color) WFD_APIEXIT;
```

```
WFD_API_CALL WFDint WFD_APIENTRY
    wfdGetPipelineLayerOrder(WFDDevice device,
                             WFDPort port,
                             WFDPipeline pipeline) WFD_APIEXIT;

#ifdef __cplusplus
}
#endif

#endif /* _WFD_H_ */
```

wfdplatform.h

```
/*********************************************************************
 *                                                                   *
 * Sample implementation of wfdplatform.h, version 1.0               *
 *                                                                   *
 * Copyright © 2008-2009 The Khronos Group                           *
 *                                                                   *
 *********************************************************************/

#ifndef _WFDPLATFORM_H_
#define _WFDPLATFORM_H_

#include <KHR/khrplatform.h>

#ifdef __cplusplus
extern "C" {
#endif

#ifndef WFD_API_CALL
#define WFD_API_CALL KHRONOS_APICALL
#endif
#ifndef WFD_APIENTRY
#define WFD_APIENTRY KHRONOS_APIENTRY
#endif
#ifndef WFD_APIEXIT
#define WFD_APIEXIT KHRONOS_APIATTRIBUTES
#endif

typedef enum
{ WFD_FALSE = KHRONOS_FALSE,
  WFD_TRUE  = KHRONOS_TRUE
} WFDboolean;

typedef khronos_uint8_t              WFDuint8;
typedef khronos_int32_t              WFDint;
typedef khronos_float_t              WFDfloat;
typedef khronos_uint32_t             WFDbitfield;
typedef khronos_uint32_t             WFDHandle;
typedef khronos_utime_nanoseconds_t WFDtime;

#define WFD_FOREVER                  (0xFFFFFFFFFFFFFFFF)

typedef void*  WFDEGLDisplay; /* An opaque handle to an EGLDisplay */
typedef void*  WFDEGLSync;    /* An opaque handle to an EGLSyncKHR */
typedef void*  WFDEGLImage;   /* An opaque handle to an EGLImage */
typedef void*  WFDNativeStreamType;

#define WFD_INVALID_SYNC             ((WFDEGLSync)0)

#ifdef __cplusplus
}
#endif

#endif /* _WFDPLATFORM_H_ */
```

10 Bibliography

EGL07 Khronos Group: *EGL_KHR_image*, Version 7, November 2007
http://www.khronos.org/registry/egl/extensions/KHR/EGL_KHR_image.txt

EGL09 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*, Version 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

11 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 (Nokia), WFC Editor
- Steven Fischer (Motorola), WFD Editor
- Lars Remes (Symbio), CT/RI Editor
- Mikko Strandborg (Acrodea)
- Samuli Lehti (Acrodea)
- Keh-Li Sheng (Aplix)
- Jarkko Kemppainen (Symbio)
- Harri Kyllönen (Symbio)
- Olli Vertanen (Symbio)
- Timo Laru (Symbio)
- 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)
- Rick Tillery (Texas Instruments)
- Tom Olsen (Texas Instruments)

12 Function Index

wfdBindMaskToPipeline 67, 68, 69, 72, 100
wfdBindPipelineToPort 52, 99
wfdBindSourceToPipeline 66, 68, 69, 72, 91, 100
wfdCreateDevice..................... 15, 16, 97
wfdCreateEvent................. 20, 23, 25, 97
wfdCreateMaskFromImage 63, 100
wfdCreateMaskFromStream 63, 100
wfdCreatePipeline................... 59, 60, 99
wfdCreatePort 32, 33, 98
wfdCreateSourceFromImage 61, 100
wfdCreateSourceFromStream..... 61, 100
wfdDestroyDevice 16, 97
wfdDestroyEvent 21, 97
wfdDestroyMask......................... 64, 100
wfdDestroyPipeline..................... 60, 100
wfdDestroyPort 33, 98
wfdDestroySource....................... 62, 100
wfdDeviceCommit 16, 18, 34, 39, 50, 52, 68, 69, 72, 76, 78, 80, 97
wfdDeviceEventAsync 24, 97
wfdDeviceEventFilter 27, 97
wfdDeviceEventWait........ 25, 26, 27, 97
wfdEnumerateDevices 14, 15, 96
wfdEnumeratePipelines 58, 59, 70, 99
wfdEnumeratePorts..... 31, 32, 41, 71, 97
wfdGetCurrentPortMode 34, 38, 39, 90, 98
wfdGetDeviceAttribi.................... 19, 97
wfdGetDisplayData....................... 54, 99
wfdGetDisplayDataFormats 53, 99
wfdGetError 10, 11, 96
wfdGetEventAttribi....................... 22, 97
wfdGetPipelineAttribf................. 78, 101
wfdGetPipelineAttribfv.............. 78, 101
wfdGetPipelineAttribi................. 78, 100
wfdGetPipelineAttribiv.............. 78, 101
wfdGetPipelineLayerOrder......... 85, 102
wfdGetPipelineTransparency...... 83, 101
wfdGetPortAttribf 11, 49, 98
wfdGetPortAttribfv 49, 98
wfdGetPortAttribi 11, 49, 98
wfdGetPortAttribiv 49, 98
wfdGetPortModeAttribf................ 37, 98
wfdGetPortModeAttribi................ 37, 98
wfdGetPortModes 36, 98
wfdGetStrings 4, 86, 87, 88, 96
wfdIsExtensionSupported 86, 89, 96
wfdSetDeviceAttribi 19, 97
wfdSetPipelineAttribf 79, 101
wfdSetPipelineAttribfv 79, 101
wfdSetPipelineAttribi 78, 79, 101
wfdSetPipelineAttribiv 79, 101
wfdSetPipelineTSColor 84, 101
wfdSetPortAttribf.......................... 51, 99
wfdSetPortAttribfv........................ 51, 99
wfdSetPortAttribi.......................... 51, 99
wfdSetPortAttribiv........................ 51, 99
wfdSetPortMode 34, 38, 39, 98