Services

3rd Party Buffer API

Filename	:	Services.3rd Party Buffer API.doc
Version	:	4.0.116 External Issue
Issue Date	:	04 Oct 2010
Author	:	PowerVR

Contents

List of Figures

Terminology

The following terminology is used in this document:

BufferClass Component	BufferClass component of the 3rd party buffer driver
OS Component	OS component of the 3rd party buffer driver
DDK	Driver Development Kit. A software package containing driver source code, allowing a specific driver to be built\modified for use on a specific platform.
PVR	PowerVR
UMA	Unified Memory Architecture – graphics device addresses system memory either allocated from the OS or reserved in the system memory
LMA	Local Memory Architecture – graphics device has its own block of memory, separate from system memory.

1. Introduction

1.1. Scope

This document specifies the interfaces for 3rd party buffer class device integration into 'version 4' of Services. An overview of buffer device class integration services component design is also provided. In addition, a 'use case' example is presented to illustrate:

- API usage
- Functionality required by 3rd party devices for services integration

1.2. Driver Integration Guidelines

This document provides a specification of the 3rd party buffer class interfaces between Services and drivers for buffer class hardware - it is not a full specification for implementing drivers for buffer class hardware.

When porting Services to a new system and/or OS it is expected that a driver will already exist for the buffer class hardware, e.g. a camera capture driver. The interface functions specified in this document should ideally be implemented by extending the interface functions of the existing hardware driver.

1.3. Buffer Class Device Types

Unlike the 3rd party display devices, buffer class device types are less easily characterised but share the follow attributes:

- 'Data Producers' – all buffer class devices are data producers (or sources) whereas display devices are data consumers (or sinks)
- 'Populate Buffers' – all buffer class devices populate one or more memory (or memory mapped) buffers with data to be imported by 'Services managed devices' via the 3rd Party Buffer API

The interfaces described in this document abstract the underlying architecture of specific buffer class hardware and provide a common control interface to varying types of buffer class devices.

1.4. Related Documents

Services Software Architectural Specification
Services Software Functional Specification
SGX DDK Porting Guide for Services 4.0
Services 3rd Party Display API

1.5. Overview of Buffer Class Architecture

Design considerations:

- Important for independent buffer class hardware to be 'coordinated' with services devices
- 3rd party Buffer Class API provides a consistent interface between services and 3rd Party Buffer device drivers
- Abstracts control of Buffer Class hardware via the Buffer Class API (used by OGLES texture stream extension)
- Provides maximum performance while maintaining the order of operations on shared resources

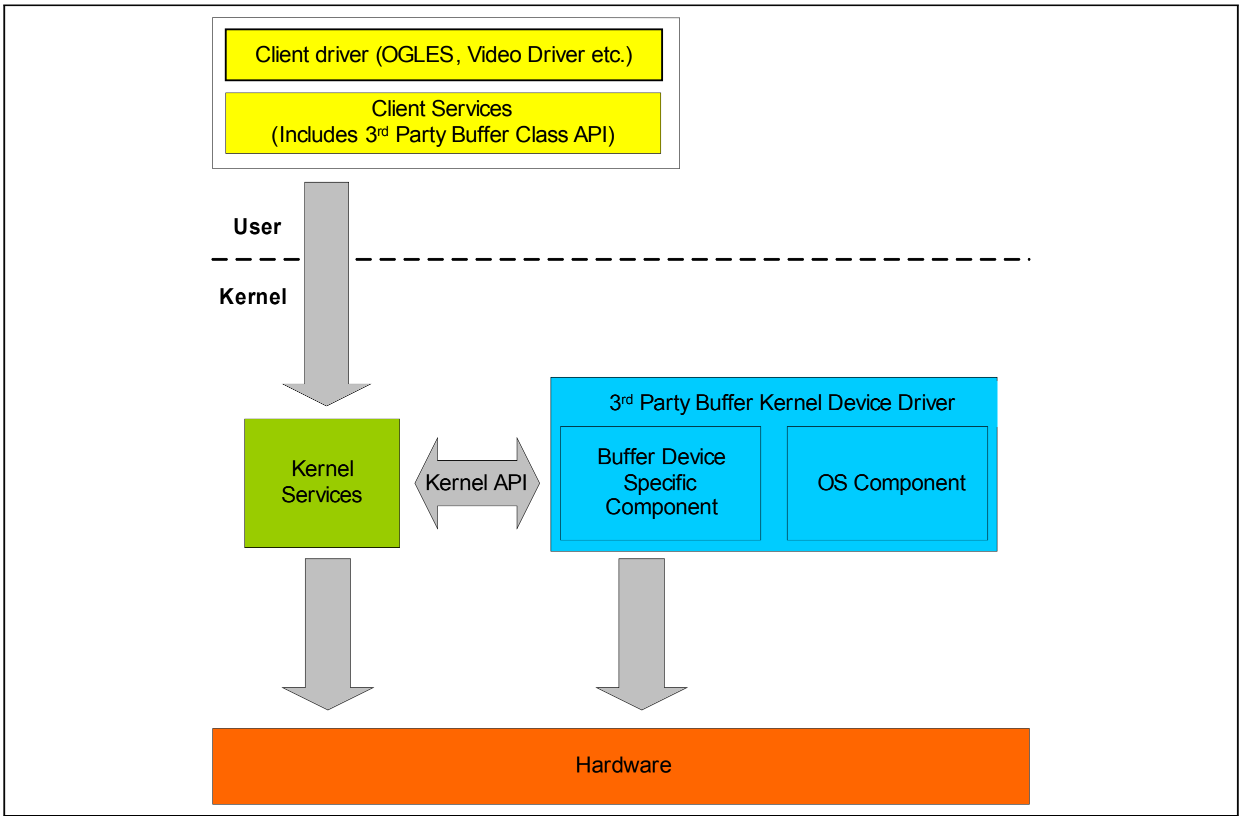

Figure 1 Buffer Class Architecture

Figure 1 highlights three distinct software components:

1. Client Driver: This directly or indirectly interfaces with a client application. It has a 'Client Services' component built into it which provides the services API and the 3rd party Buffer Class API.
2. Kernel Services: The 'kernel mode' Services component
3. 3rd Party Buffer Kernel Device Driver: This is a kernel device driver for controlling the Buffer Class hardware. 'Interfacing code' is added to the driver allowing the buffer device to be integrated with other Services managed devices. There are two sub components:
    - Buffer Class Device Specific Component: This contains buffer device specific code to implement callback functions from Kernel Services.
    - OS Component: This is component provides OS abstraction functions

The Arrows in the diagram represent calls through interfaces between software and hardware components. The labelled arrow, 'Kernel API' is described in the following sections.

1.6. SGX and Buffer Class Device Addressable Surfaces

The 3rd Party buffer class API specification provides support for direct access to buffer class device surfaces by SGX devices. SGX devices can read from buffer class device addressable surfaces therefore the buffer class device surfaces attributes are limited by the constraints of both the buffer class and SGX devices.

The primary difference between many buffer class devices and SGX is the surface stride requirements. See Appendix C below for details.

2. The 3rd Party Private Device Data Structure

The 3rd party private device data structure is usually called XXX_DEVINFO, with the 'XXX' being a reference to the 3rd party buffer hardware. XXX_DEVINFO is allocated and initialized by the 3rd party driver's 'Init' function.

Within the 3rd party driver, a global pointer is used to access XXX_DEVINFO (i.e. a static variable declared within the BufferClass component).

OpenBCDevice (a services-to-buffer API) returns an XXX_DEVINFO handle to kernel services. Kernel Services retains a copy of this handle and uses it as an argument to other services-to-buffer APIs.

Note: XXX_DEVINFO can only be accessed within the 3rd party driver.

Here is an example XXX_DEVINFO definition:

```
typedef struct XXX_DEVINFO_TAG
{
	/* device id assigned by services */
	IMG_UINT32			ui32DeviceID;

	/* array of references to buffers */
	XXX_BUFFER			*psSystemBuffer;

	/* common buffer information */
	BUFFER_INFO			sBufferInfo;

	/* number of supported buffers */
	IMG_UINT32			ui32NumBuffers;

	/* jump table into PVR services */
	PVRSRV_BC_BUFFER2SRV_KMJTABLE sPVRJTable;

	/* jump table into BC */
	PVRSRV_BC_SRV2BUFFER_KMJTABLE sBCJTable;

	/* handle for connection to kernel services */
	IMG_HANDLE			hPVRServices;

	/* ref count */
	IMG_UINT32			ui32RefCount;

}  XXX_DEVINFO;
```

3. The 3rd Party Kernel Driver initialisation

This section describes the 3rd party kernel driver initialisation and de-initialisation functions. The API is not fixed and the specific implementation is the decision of the 3rd party driver writer. However, the descriptions serve as a template on how to structure the initialisation of a 3rd party kernel driver.

Initialisation is expected to occur at 3rd party kernel driver load time. It is important that the 3rd party driver is loaded after kernel services.

Init

```
PVRSRV_ERROR Init();
```

Inputs

Outputs

Returns

`PVRSRV_ERROR_OUT_OF_MEMORY`	Memory allocation failure
`PVRSRV_ERROR_INIT_FAILURE`	Initialisation failure
`PVRSRV_ERROR_DEVICE_REGISTER_FAILED`	Failed to register device with services
`PVRSRV_OK`	Success

Component

Implemented in the BufferClass component of the 3rd party kernel driver

Description

Allocates and sets up the device's private data structure (i.e. XXX_DEVINFO).

Initiates a connection to kernel services.

Acquires the kernel services Buffer Class jump table, enabling calls from the 3rd party device into kernel services.

Sets up its own Buffer Class jump table, enabling calls from kernel services into the 3rd party driver.

Registers the device as a buffer class device type with kernel services.

A pointer to the XXX_DEVINFO, the device's private data structure, is stored in a static variable.

Deinit

```
PVRSRV_ERROR Deinit();
```

Inputs

Outputs

Returns

`PVRSRV_OK`	Success
`PVRSRV_ERROR_GENERIC`	Fail

Component

Implemented in the BufferClass component of the 3rd party kernel driver

Description

This function is called when the 3rd party kernel driver is unloaded. It decrements the reference count and will perform the following de-initialisation tasks if the reference count is equal to zero:

1. Remove the device from the services buffer device class list, de-allocating any associated resources
2. Closes the connection to kernel services
3. De-allocates any resources in the 3rd party kernel driver.

4. ‘OS Component’ Functions

This section outlines some of the functions that should be implemented in the OS component of the 3rd party buffer driver. Note: the API is not fixed and the specific implementation is the decision of the 3rd party driver writer.

OpenPVRServices

```
PVRSRV_ERROR OpenPVRServices (IMG_HANDLE *phPVRServices);
```

Outputs

phPVRServices	Handle for connection from 3rd party kernel driver to kernel services

Returns

PVRSRV_OK	Success
PVRSRV_ERROR_INVALID_DEVICE	Fail

Component

Implemented in the OS component of the 3rd party kernel driver

Description

This function opens a connection from a 3rd party kernel driver to the kernel services. This must be called before any other kernel APIs in this section.

Note: depending on the Operating System, this API may or may not be required

ClosePVRServices

```
PVRSRV_ERROR ClosePVRServices (IMG_HANDLE hPVRServices);
```

Inputs

hPVRServices	Handle for connection from 3rd party kernel driver to kernel services

Outputs

Returns

`PVRSRV_OK`	Success

Component

Implemented in the OS component of the 3rd party kernel driver

Description

This function closes a connection from a 3rd party kernel driver to the kernel services. It is implemented in the OS component 3rd party kernel driver.

Note: depending on the Operating System, this API may or may not be required

5. Kernel API

The ‘Kernel API’ arrow (shown in Figure 1) describes the connection between kernel services and the 3rd party buffer class driver. More specifically, the arrow represents two interfaces each flowing in opposing directions:

1. Buffer Device-to-Services: APIs called from within the 3rd party buffer driver and implemented in kernel services, see ‘Buffer Device to Services’ Kernel API.
2. Services-to-Buffer Device: APIs called from within kernel services and implemented in the 3rd party Buffer Class driver, see ‘Services to Buffer Device’ Kernel API.

5.1. ‘Buffer Device to Services’ Kernel API

The BufferClass component contains the ‘consumer services knowledge’ that must be built into the 3rd party kernel driver in order to interface with kernel services directly.

The APIs described here are retrieved from kernel services via a data structure of function pointers. Kernel services exports the function `PVRGetBufferClassJTable` which the 3rd party buffer driver must call in order to acquire the function pointers.

PVRGetBufferClassJTable

```
IMG_EXPORT IMG_BOOL PVRGetBufferClassJTable(PVRSRV_BC_BUFFER2SRV_KMJTABLE
              *psJTable);
```

Inputs

psJTable	Structure of function pointers each corresponding to a kernel services API function

Outputs

Returns

IMG_TRUE	Success
IMG_FALSE	Fail

Component

Implemented in kernel services

Description

This function is used by the 3rd party buffer driver to retrieve kernel services API functions via a pre-defined structure of function pointers. Each API is documented in the following sections.

For clarity the function pointer structure definition is also presented:

```
/* Function table for BUFFER->SRVKM */
typedef struct PVRSRV_BC_BUFFER2SRV_KMJTABLE_TAG
{
	IMG_UINT32                        ui32TableSize;
	PFN_BC_REGISTER_BUFFER_DEV        pfnPVRSRVRegisterBCDevice;
	PFN_BC_REMOVE_BUFFER_DEV          pfnPVRSRVRemoveBCDevice;

} PVRSRV_BC_BUFFER2SRV_KMJTABLE, *PPVRSRV_BC_BUFFER2SRV_KMJTABLE;
```

Note: The 3rd party buffer driver must first acquire a pointer to PVRGetBufferClassJTable by calling into the OS component (different OS environments will do this in different ways).

PVRSRVRegisterBCDeviceKM

```
PVRSRV_ERROR PVRSRVRegisterBCDeviceKM (PVRSRV_BC_SRV2BUFFER_KMJTABLE
              *psFuncTable, IMG_UINT32 *pui32DeviceID );
```

Inputs

psFuncTable	Function table for Srvkm->Buffer

Outputs

pui32DeviceID	Unique identifier index allocated to the 3rd party device

Returns

`PVRSRV_OK`	Success
`PVRSRV_ERROR_OUT_OF_MEMORY`	Memory allocation failure
`PVRSRV_ERROR_GENERIC`	Fail

Component

Implemented in kernel services – eurasia\services\srvkm\common\deviceclass.c

Access by buffer device-to-services function pointer

pfnPVRSRVRegisterBCDevice

Description

This function registers the 3rd party device with kernel services (the 3rd party device registers its ‘services-to-buffer’ function table with kernel services). A device node is allocated and added to the device node list.

The device node has the following attributes and data:

- Device Type: PVRSRV_DEVICE_TYPE_EXT
- Device Class: PVRSRV_DEVICE_CLASS_BUFFER
- A reference count
- A ‘srvkm-to-buffer device’ jump-table. This mechanism permits 3rd party buffer driver functionality to be invoked by kernel services. Thus the 3rd party device can be controlled via the buffer class APIs within client services.

PVRSRVRemoveBCDeviceKM

```
PVRSRV_ERROR PVRSRVRemoveBCDeviceKM(IMG_UINT32 ui32DevIndex);
```

Inputs

ui32DevIndex	Unique device identifier representing the device to remove from consumer services control.

Outputs

Returns

`PVRSRV_OK`	Success
`PVRSRV_ERROR_GENERIC`	Fail

Component

Implemented in kernel services – eurasia\services\srvkm\common\deviceclass.c

Access by buffer device-to-services function pointer

pfnPVRSRVRemoveBCDevice

Description

This function removes 'control' of a specified 3rd party device from kernel services. 'Device Removal' entails:

- Using the device index supplied by the caller to locate the device node to be removed
- Deleting the device from the device class linked-list so that it cannot be controlled via the Buffer class APIs.
- De-allocating all data structures allocated by PVRSRVRegisterBCDeviceKM, including the 'srvkm-to-buffer device' jump-table.
- De-allocating 'synchronisation object' temporary storage

5.2. 'Services to Buffer Device' Kernel API

These APIs are implemented in the 3rd party buffer device driver and are accessed as follows:

- The client driver (e.g. OPENGLES) calls Client Services Buffer Class APIs
- Client Services Buffer Class APIs are routed through to Kernel Services via the client/kernel glue layer
- Kernel Services will invoke BufferClass APIs by using the 'services-to-buffer device' jump-table – through which 3rd party buffer device can be controlled.

The 3rd party buffer device driver uses PVRSRVRegisterBCDeviceKM to pass the jump-table data to kernel services. Here is the jump-table structure:

```
/* Function table for SRVKM->BUFFER */
typedef struct PVRSRV_BC_SRV2BUFFER_KMJTABLE_TAG
{
	IMG_UINT32			ui32TableSize;
	PFN_OPEN_BC_DEVICE		pfnOpenBCDevice;
	PFN_CLOSE_BC_DEVICE		pfnCloseBCDevice;
	PFN_GET_BC_INFO			pfnGetBCInfo;
	PFN_GET_BC_BUFFER		pfnGetBCBuffer;
	PFN_GET_BUFFER_ADDR		pfnGetBufferAddr;

} PVRSRV_BC_SRV2BUFFER_KMJTABLE;
```

OpenBCDevice

```
static PVRSRV_ERROR OpenBCDevice(IMG_UINT32 ui32DeviceID, IMG_HANDLE
               *phDevice);
```

Inputs	
ui32DeviceID	ID of the device to open

Outputs	
phDevice	Handle to device info structure (XXX_DEVINFO)

Returns	
PVRSRV_OK	Success
PVRSRV_ERROR_GENERIC	Fail

Component

Implemented in the BufferClass component of the 3rd party driver – XXX_bufferclass.c

Access by services-to-buffer device function pointer

pfnOpenBCDevice

Description

Stores the system surface sync data in the device’s private data structure (XXX_DEVINFO) and returns a handle to XXX_DEVINFO.

CloseBCDevice

```
static PVRSRV_ERROR CloseBCDevice(IMG_UINT32 ui32DeviceID, IMG_HANDLE
               hDevice);
```

Inputs

`ui32DeviceID`	ID of the device to close
`hDevice`	Handle to device's private data structure

Outputs

Returns

`PVRSRV_OK`	Success
`PVRSRV_ERROR_GENERIC`	Fail

Component

Implemented in the BufferClass component of the 3rd party driver – XXX_bufferclass.c

Access by services-to-buffer device function pointer

pfnCloseBCDevice

Description

Close connection to the buffer class device.

GetBCInfo

```
static PVRSRV_ERROR GetBCInfo(IMG_HANDLE hDevice, BUFFER_INFO *psBCInfo);
```

Inputs

hDevice	Handle to 3rd party private data – XXX_DEVINFO

Outputs

psBCInfo	Handle to system buffer (primary surface)

Returns

PVRSRV_OK	Success
PVRSRV_ERROR_GENERIC	Fail

Component

Implemented in the BufferClass component of the 3rd party driver

Access by services-to-buffer device function pointer

pfnGetBCInfo

Description

This function returns common buffer information.

```
/* common buffer information structure */
typedef struct BUFFER_INFO_TAG
{
	/* buffer count */
	IMG_UINT32			ui32BufferCount;
	/* buffer device ID */
	IMG_UINT32			ui32BufferDeviceID;
	/* pixel format */
	PVRSRV_PIXEL_FORMAT		pixelformat;
	/* surface stride */
	IMG_UINT32			ui32ByteStride;
	/* pixel width */
	IMG_UINT32			ui32Width;
	/* pixel height */
	IMG_UINT32			ui32Height;
	/* flags */
	IMG_UINT32			ui32Flags;
} BUFFER_INFO;

#define PVRSRV_BC_FLAGS_YUVCSC_CONFORMANT_RANGE	(0 << 0)
#define PVRSRV_BC_FLAGS_YUVCSC_FULL_RANGE		(1 << 0)

#define PVRSRV_BC_FLAGS_YUVCSC_BT601			(0 << 1)
#define PVRSRV_BC_FLAGS_YUVCSC_BT702			(1 << 1)
```

GetBCBuffer

```
static PVRSRV_ERROR GetBCBuffer(IMG_HANDLE hDevice, IMG_UINT32
                ui32BufferNumber, PVRSRV_SYNC_DATA *psSyncData, IMG_HANDLE
                *phBuffer);
```

Inputs

hDevice	Handle to 3rd party private data – XXX_DEVINFO
ui32BufferNumber	Buffer number to return
psSyncData	Pointer to syncdata item

Outputs

phBuffer	Pointer to buffer handle

Returns

PVRSRV_OK	Success
PVRSRV_ERROR_INVALID_PARAMS	Fail

Component

Implemented in the BufferClass component of the 3rd party driver

Access by services-to-buffer device function pointer

pfnGetBCBuffer

Description

This function returns a handle to a buffer associated with ui32BufferCount.

GetBCBufferAddr

```
static PVRSRV_ERROR GetBCBufferAddr(IMG_HANDLE hDevice, IMG_HANDLE
               hBuffer, IMG_SYS_PHYADDR **ppsSysAddr, IMG_UINT32
               *pui32ByteSize, IMG_VOID **ppvCpuVAddr, IMG_HANDLE
               *phOSMapInfo, IMG_BOOL *pbIsContiguous, IMG_UINT32
               *pui32TilingStride);
```

Inputs	
hDevice	Handle to 3rd party private data – XXX_DEVINFO
hBuffer	Handle to buffer – XXX_BUFFER

Outputs	
ppsSysAddr	System physical address of system buffer
pui32ByteSize	Size of system buffer in bytes
ppvCpuVAddr	CPU virtual address of system buffer
phOSMapInfo	(optional) an OS Mapping handle for KM->UM surface mapping
pbIsContiguous	Indicates whether system is buffer made up of contiguous pages
pui32TilingStride	Tile stride of surface (if memory tiling is supported by device)

Returns	
`PVRSRV_OK`	Success
`PVRSRV_ERROR_INVALID_PARAMS`	Fail

Component

Implemented in the BufferClass component of the 3rd party driver

Access by services-to-buffer device function pointer

pfnGetBufferAddr

Description

This function provides the memory mapping information for a buffer. The caller must supply a handle to the data structure containing buffer information (i.e. a handle to XXX_BUFFER). The following information is returned:

- The system physical address of the system buffer
- CPU virtual addresses of the system buffer
- The size of the system buffer in bytes
- A Boolean value to indicate whether the buffer memory consists of contiguous or non-contiguous pages.

OS Mapping handle for KM->UM surface mapping – this is optional and should be set to NULL if unused.

6. Client Services Buffer Class API

Client drivers (e.g. OGLES) control buffer class device(s) by calling the Client Services Buffer Class APIs. This set of functions route through to Kernel Services and, where appropriate, are despatched to the 3rd Party Buffer driver via a table of function pointers. This section describes the APIs presented to the client drivers.

PVRSRVEnumerateDeviceClass

```
PVRSRV_ERROR PVRSRVEnumerateDeviceClass(PVRSRV_CONNECTION *psConnection,
               PVRSRV_DEVICE_CLASS DeviceClass, IMG_UINT32 *pui32DevCount,
               IMG_UINT32 *pui32DevID );
```

Inputs

psConnection	Bridge Connection information
DeviceClass	Device Class Type (Buffer in this case)

Outputs

pui32DevCount	Number of devices present
pui32DevID	Pointer to an array of Device IDs for each device

Returns

PVRSRV_OK	Success
PVRSRV_ERROR_GENERIC	Fail

Component

Implemented in the Services Client component

Description

This function enumerates 'Device Class' type devices. It is generally called in two phases:

1. pui32DevID==NULL and pui32DevCount==valid pointer. pui32DevCount returns the number of devices in the system
2. Use value returned to allocate an array of IDs and pass address of array in pui32DevID. Valid list of IDs is copied into the array.

The driver can choose from one or more devices based on the ID, 'opening' a given device by passing the correct ID.

PVRSRVOpenBCDevice

```
IMG_HANDLE PVRSRVOpenBCDevice(PVRSRV_DEV_DATA *psDevData, IMG_UINT32
               ui32DeviceID);
```

Inputs

psDevData	Device data information
ui32DeviceID	ID of the device to open

Outputs

Returns

`Valid handle to the device`	Success
`NULL`	Fail

Component

Implemented in the Services Client component

Description

This function ‘opens’ a given buffer class device specified by the device’s ID.

PVRSRVCloseBCDevice

```
PVRSRV_ERROR PVRSRVCloseBCDevice(PVRSRV_CONNECTION    *psConnection,
               IMG_HANDLE hDevice);
```

Inputs

psConnection	Bridge Connection information
hDevice	Handle for the device

Outputs

Returns

PVRSRV_OK	Success
PVRSRV_ERROR_GENERIC	Fail

Component

Implemented in the Services Client component

Description

This function ‘closes’ a given buffer device specified by the device’s handle.

PVRSRVGetBCBufferInfo

```
PVRSRV_ERROR PVRSRVGetBCBufferInfo (IMG_HANDLE hDevice, BUFFER_INFO
              *psBuffer);
```

Inputs

hDevice	Handle to 3rd party private data – XXX_DEVINFO

Outputs

psBuffer	Common buffer information

Returns

PVRSRV_OK	Success
PVRSRV_ERROR_GENERIC	Fail

Component

Implemented in the Services Client component

Description

This function returns common buffer information.

```
/* common buffer information structure */
typedef struct BUFFER_INFO_TAG
{
	/* buffer count */
	IMG_UINT32			ui32BufferCount;
	/* buffer device ID */
	IMG_UINT32			ui32BufferDeviceID;
	/* pixel format */
	PVRSRV_PIXEL_FORMAT		pixelformat;
	/* surface stride */
	IMG_UINT32			ui32ByteStride;
	/* pixel width */
	IMG_UINT32			ui32Width;
	/* pixel height */
	IMG_UINT32			ui32Height;
	/* OEM specific flags */
	IMG_UINT32			ui32OEMFlags;
} BUFFER_INFO;
```

PVRSRVGetBCBuffer

```
static PVRSRV_ERROR PVRSRVGetBCBuffer (IMG_HANDLE hDevice, IMG_UINT32
               ui32BufferIndex, IMG_HANDLE *phBuffer);
```

Inputs

hDevice	Handle to 3rd party private data – XXX_DEVINFO
ui32BufferIndex	Buffer index to return handle for

Outputs

phBuffer	Pointer to buffer handle

Returns

`PVRSRV_OK`	Success
`PVRSRV_ERROR_INVALID_PARAMS`	Fail

Component

Implemented in the Services Client component

Description

This function returns a handle to a buffer associated with ui32BufferCount.

7. Use Case Example

This section presents a ‘use case’ example in which a ‘3rd party buffer device’ and its device driver are integrated into the Consumer Services and controlled via the Buffer Class APIs.

7.1. Dynamic Initialisation

Kernel Services has no knowledge of the 3rd party devices in the system until they connect to services. The 3rd party buffer class kernel device driver must unconditionally register with kernel services at initialisation.

After registration, the client driver will be able to call PVRSRVEnumerateDeviceClass and PVRSRVOpenBCDevice

Note: the ‘software component’ responsible for causing the 3rd party buffer class device to register with kernel services may vary depending on the system and/or environment, e.g. in WinXP the entry point for the 3rd party driver is: DriverEntry (which will call Init).

7.1.1. Initialisation sequence

This example describes the call sequence of a client driver initialising the buffer class device using the 3rd party buffer class API. Other kernel services APIs used by the client driver are not considered here. Initialisation generally starts at driver load time (note: the 3rd party kernel driver is loaded after kernel services). The 3rd party driver does the following at initialisation:

1. Allocates and sets up the device’s private data structure
2. Initiates a connection to kernel services by calling OpenPVRServices
3. Acquires the kernel services Buffer Class jump table, enabling calls from the 3rd party device into kernel services.
4. Sets up its own Buffer Class jump table, enabling calls from kernel services into the 3rd party driver.
5. Registers the device as a buffer class device type with kernel services by calling pfnPVRSRVRegisterBCDevice.

The client driver makes the following calls:

PVRSRVEnumerateDeviceClass

Called twice: the first time to enumerate the buffer devices available, and the second time to get their device IDs.

PVRSRVOpenBCDevice

Called once and does the following:

1. Finds the matching device node – i.e. matching device class and device ID
2. Calls into the 3rd party driver via pfnOpenBCDevice to acquire a handle to the device’s private data structure.
3. Returns a handle to buffer device management structure, which contains the handle to the 3rd party devices private data structure.

Now that the client driver has a handle to the buffer device it can call other buffer class functions. Here are some of the other buffer class APIs that may be called:

PVRSRVGetBCBufferInfo

Returns a pointer to the 3rd party buffer class driver’s BUFFER_INFO structure

PVRSRVGetBCBuffer

Gets buffer handle by index

PVRSRVMapDeviceClassMemory

Map Buffer Class buffers to Services managed devices

Once a client driver has buffer device buffer mapping(s) on other Services managed devices the buffers can be read synchronously or asynchronously. The sequencing of read (Services device) operations and write (Buffer Device) operations is managed by the synchronisation object that services associates with each buffer device buffer.

Appendix A. Texture Stream Extension (GL_IMG_texture_stream)

```
Name

    IMG_texture_stream

Name Strings

    GL_IMG_texture_stream

Notice

    Copyright Imagination Technologies Limited, 2005.

Contact

    Graham Connor, Imagination Technologies (graham 'dot' connor 'at'
    powervr 'dot' com)

Status

    DRAFT

Version

    Draft 0.5, 5 March 2007

Number

    XXX

Dependencies

    OpenGL ES 1.0 is required.

    This extension is written against the OpenGL ES 1.0 Specification, (which in turn
    is a derived from OpenGL 1.3). Thus this spec is effectively written against OpenGL
    1.3 but does not address sections explicitly removed or reduced by OpenGL-ES 1.0.

Overview

    This extension is designed to allow OpenGL to use directly a hardware source which
    contains image data as a texture. Specifically this extension is designed to avoid
    any copying of texture data from this source, and does so in a manner which avoids
    overloading in any manner any existing texture functionality where a pointer is
    passed in to the GL as the source data. This extension does not attempt to address
    synchronisation of the use of the hardware source w.r.t when data is fetched for
    fragment generation. Such synchronisation can be addressed in a further extension.

Issues

    1) Does this extension support multiple independant sources of data?

    Yes, through the device argument. The number of valid devices in a system
    can be queried. However the type of device and order of the devices are platform
    dependant.

    2) Does this extension support multiple textures from one source device?

    Yes, through the deviceoffset argument which allows the image to be used as a texture
    to be offset from the base of the device.

    3) Could this extension be used to source live video data?

    Yes, but sysnchronisation would have to be addressed. Conceivably the video could
    be multi-buffered and the deviceoffset used to bind different texture objects to the
    buffers.

    4) Who initialises/owns the underlying device?

    Not GL. GL can get the device attibutes and return then to the application so that
    it is aware of the underlying format and size of the device.
```

```
    5) What is the deviceoffset argument to TexBindStream?

    This moves the texture source a whole device image along. This allows support for
    multibuffering.

    6) Could this extension be used to allow applications other than
       HW drivers to provide a source of texture data?

    Yes, but such applications would need guidlines and support beyond
    the scope of this extension to provide the necessary framework
    required. This is likely to be beyond the scope of casual developers.

New Procedures and Functions

    void TexBindStreamIMG(GLint device,
                          GLint deviceoffset);

    void GetTexStreamDeviceAttributeivIMG(GLint device,
                                          GLenum pname,
                                          GLint *params);

    const GLubyte * GetTexStreamDeviceNameIMG(GLint device);

New Tokens

    Accepted by the <cap> parameter of Enable, Disable, and by the <target> parameter of
    BindTexture:

        TEXTURE_STREAM_IMG                         0x8C0D

    Accepted by the <pname> parameter of GetIntegerv, and GetFixedv:

        TEXTURE_NUM_STREAM_DEVICES_IMG             0x8C0E

    Accepted by the <pname> parameter of GetTexStreamDeviceAttributeivIMG:

        TEXTURE_STREAM_DEVICE_WIDTH_IMG                    0x8C0F
        TEXTURE_STREAM_DEVICE_HEIGHT_IMG        0x8EA0
        TEXTURE_STREAM_DEVICE_FORMAT_IMG        0x8EA1
        TEXTURE_STREAM_DEVICE_NUM_BUFFERS_IMG   0x8EA2

Errors

    INVALID_VALUE is set if <TBD>

Example Usage

    /* ===============================================
     * Example 1 - Simple usage a single layer texture
     * This example would apply in the case where the
     * stream device is a camera which is optically
     * viewfound and a single snap shot is taken and
     * displayed after the shot for review.
     * ===============================================/
    /*
     * Put texture machine into camera texture mode
     */
    glBindTexture(GL_TEXTURE_STREAM_IMG, TexName);
    /*
     * Bind a specific location of texture camera device 0
     * to a QVGA resolution texture
     */
    glTexBindStreamIMG(0, 0);
    glEnable(GL_TEXTURE_STREAM);
    /*
     * Draw object here ...
     */


    /* ====================================================
     * Example 2 - Quering the devices on the system
     * ====================================================/
    /*
     * Find out how many devices are on the system
     */
```

```
    glGetIntegerv(GL_TEXTURE_NUM_STREAM_DEVICES_IMG, &iNumDevices);
    for(i=0; i<nNumDevices; i++)
    {
        /*
         * Query each device
         */
        glGetTexStreamDeviceParameterivIMG(i, GL_TEXTURE_STREAM_DEVICE_WIDTH_IMG &iWidth);
        glGetTexStreamDeviceParameterivIMG(i, GL_TEXTURE_STREAM_DEVICE_HEIGHT_IMG &iHeight);
        glGetTexStreamDeviceParameterivIMG(i, GL_TEXTURE_STREAM_DEVICE_FORMAT_IMG &iFormat);
        glGetTexStreamDeviceParameterivIMG(i, GL_TEXTURE_STREAM_DEVICE_NUM_BUFFERS_IMG
&iBuffers);
    }

    /* ===============================================
     * Example 3 - Binding N buffers to N textures
     * This example would apply where an LCD is used instead of
     * an optical viewfinder and the LCD continously displays
     * the CCD image
     * ===============================================/
    glGetIntegerv(GL_TEXTURE_NUM_STREAM_DEVICES_IMG, &iNumDevices);

      /*
     * Identify the device wanted
     */
    for(i=0; i<nNumDevices; i++)
    {
       GLubyte *name;
       /*
        * Query each device name
        */
       name = glGetTexStreamDeviceNameIMG(i);

       if(IsThisRequiredDevice(name))
       {
           device = i;
         break;
       }
    }

    glGetTexStreamDeviceParameterivIMG(device, GL_TEXTURE_STREAM_DEVICE_NUM_BUFFERS_IMG
&iBuffers);

      glGetTextures(iBuffers, &TexNames);

      for(i=0; i<iBuffers; i++)
    {
      /*
       * Put texture machine into stream texture mode
       */
       glBindTexture(GL_TEXTURE_STREAM_IMG, TexNames[i]);
      /*
       * Bind a specific buffer of texture camera device 0
       */
       glTexBindStreamIMG(device, i);
    }

      frame=0
    glEnable(GL_TEXTURE_STREAM);
    /*
     * Main Draw Loop for free running viewfinder
     */
      do
    {
       /*
        * Host Camera API stuff
        */
       CapturePhotoToBuffer(frame%iBuffers);

       glBindTexture(GL_TEXTURE_STREAM_IMG, frame%iBuffers);
       /*
        * Draw object here ...
        */
       eglSwapBuffers();
    } while(no_exit)

Revision History
```

```
    0.1, 30/4/2004 gdc: Initial revision.
    0.2, 8/9/2004 gdc: First released draft.
    0.3, 19/1/2005 gdc: Renamed to texture_stream.
       0.4, 7/2/2007 jml: Corrected example usage.
       0.5, 5/3/2007 jml: Updated enumerants.
```

Appendix B. Texture Stream Extension for OpenGL ES 2.0 (GL_IMG_texture_stream2)

```
Name

    IMG_texture_stream2

Name Strings

    GL_IMG_texture_stream2

Contributors

    Ben Bowman

Contact

    Graham Connor, Imagination Technologies (graham 'dot' connor 'at'
    powervr 'dot' com)

Notice

    Copyright © 2008 Imagination Technologies

IP Status

    None.

Status

    Draft

Version

    Draft 0.3, Last Modified Date: November 24, 2008

Number

    XXX

Dependencies

    OpenGL ES 2.0 is required.
    This extension is written against the OpenGL ES 2.0 full
    specification (revision 2.0.23) and the OpenGL ES Shading
    Language version 1.00 (revision 1.0.16)

Overview

    This extension is designed to allow OpenGL to use directly a hardware source which
    contains image data as a texture. Specifically this extension is designed to avoid
    any copying of texture data from this source, and does so in a manner which avoids
    overloading in any manner any existing texture functionality where a pointer is
    passed in to the GL as the source data. This extension does not attempt to address
    synchronisation of the use of the hardware source w.r.t when data is fetched for
    fragment generation. Such synchronisation can be addressed in a further extension.


Issues

    1) Does this extension support multiple independent sources of data?

    Yes, through the device argument. The number of valid devices in a system
    can be queried. However the type of device and order of the devices are platform
    dependant.

    2) Does this extension support multiple textures from one source device?

    Yes, through the deviceoffset argument which allows the image to be used as a texture
    to be offset from the base of the device.

    3) Could this extension be used to source live video data?
```

```
    Yes, but synchronisation would have to be addressed. Conceivably the video could
    be multi-buffered and the deviceoffset used to bind different texture objects to the
    buffers.

    4) Who initialises/owns the underlying device?

    Not GL. GL can get the device attributes and return then to the application so that
    it is aware of the underlying format and size of the device.

    5) What is the deviceoffset argument to TexBindStream?

    This moves the texture source a whole device image along. This allows support for
    multibuffering.

    6) Could this extension be used to allow applications other than
       HW drivers to provide a source of texture data?

    Yes, but such applications would need guidelines and support beyond
    the scope of this extension to provide the necessary framework
    required. This is likely to be beyond the scope of casual developers.

    7) Is a new texture target necessary, or could GL_TEXTURE_2D be used?

    The main issue with overloading the texture 2d target is that the GL needs a way to map
the
    stream device and offset onto the 2D. This seems difficult without an explicit enable,
    which has been removed from OpenGL ES 2.0 in favour of using the sampler type. Adding an
    enable, but using the 2D target would seem to break the goal of no overloading of
existing
    texture functionality. This has the benefit of matching the interface in the original
    IMG_texture_stream extension.

    8) Is a new sampler type and lookup function necessary, or could sampler2D be used?

    Given the resolution to issue 7 above, it seems sensible to create a new sampler, rather
than
    attempting to overload sampler2D. This has the benefit of allowing the compiler to
detect any
    special work which may need to be done to support samplers of samplerStream type which
have
    implicit format conversion in the lookup. This seems to match the decisions made for
shadow
    samplers in GLSL in the past.

    9) Should the result of a textureStream lookup be in the form R,G,B,A?

    Following from the original IMG_texture_stream extension, it would seem sensible to
return
    RGBA from the sampler, and leave all format conversion to the compiler, rather than
requiring
    the shader writer to perform (for example) color-space conversion in (potentially hard
to
    optimise) shader code.


New Tokens

    Accepted by the <target> parameter of BindTexture, TexParameter[i/f/x][v],
    GetTexParameteriv, and GetTexParameterfv:

        TEXTURE_STREAM_IMG                      0x8C0D

    Accepted by the <pname> parameter of GetIntegerv, and GetFixedv:

        TEXTURE_NUM_STREAM_DEVICES_IMG          0x8C0E
        TEXTURE_BINDING_STREAM_IMG              ????

    Accepted by the <pname> parameter of GetTexStreamDeviceAttributeivIMG:

        TEXTURE_STREAM_DEVICE_WIDTH_IMG         0x8C0F
        TEXTURE_STREAM_DEVICE_HEIGHT_IMG        0x8EA0
        TEXTURE_STREAM_DEVICE_FORMAT_IMG        0x8EA1
        TEXTURE_STREAM_DEVICE_NUM_BUFFERS_IMG   0x8EA2

       Returned by the <type> parameter of GetActiveUniform:
              SAMPLER_STREAM_IMG                                            ????
```

```
New Procedures and Functions

    void TexBindStreamIMG(GLint device,
                          GLint deviceoffset);

    void GetTexStreamDeviceAttributeivIMG(GLint device,
                                          GLenum pname,
                                          GLint *params);

    const GLubyte * GetTexStreamDeviceNameIMG(GLint device);

Additions to Chapter 2 of the OpenGL ES 2.0 Specification (OpenGL ES Operation)

   Modify Section 2.10.4, Shader Variables, p. 30
       (Modify the paragraph beginning "For the selected uniform, the type of the
        uniform is returned into type" on p. 35)
              For the selected uniform, the type of the uniform is returned into type
              size of the uniform is returned into size. The value in size is in units of
the
              type returned in type. The type returned can be any of FLOAT, FLOAT_VEC2,
              FLOAT_VEC3, FLOAT_VEC4, INT, INT_VEC2, INT_VEC3, INT_VEC4, BOOL,
              BOOL_VEC2, BOOL_VEC3, BOOL_VEC4, FLOAT_MAT2, FLOAT_MAT3, FLOAT_MAT4,
              SAMPLER_2D, SAMPLER_CUBE, or SAMPLER_STREAM_IMG.

Additions to Chapter 3 of the OpenGL ES 2.0 Specification (Rasterisation)

   Modify Section 3.7.4, Texture Parameters, p. 73
       (Modify the sentence after the TexParameter{if}v prototype, p. 73)
       <target> is the target, which must be TEXTURE_2D, TEXTURE_CUBE_MAP, or
       TEXTURE_STREAM_IMG.

   Modify Section 3.7.10, Texture Completeness and Non-Power-Of-Two Textures, p. 80
    (Add a paragraph to the section on mipmap completeness on p. 81)
     ...
     * Each dimension of the zero level array is positive.

     For stream textures, a texture is inherently complete as mip-mapping is not supported.

     For cube map textures, a texture is cube complete if the following conditions
       all hold true:
       ...

   Modify Section 3.7.13, Texture Objects, p. 82
       (Modify the paragraphs in section 3.7.13, p. 82)
       ...
       In addition to the default textures TEXTURE_2D, TEXTURE_CUBE_MAP and
TEXTURE_STREAM_IMG,  named two-dimensional, cube map and stream texture objects can be
created and operated upon.
       The name space for texture objects is the unsigned integers, with zero reserved by
       the GL.
       A texture object is created by binding an unused name to TEXTURE_2D,
TEXTURE_CUBE_MAP, or
       TEXTURE_STREAM_IMG.
       ...
       If the new texture object is bound to TEXTURE_2D, TEXTURE_CUBE_MAP, or
TEXTURE_STREAM_IMG,
       it is and remains a two-dimensional, cube map, or stream texture respectively until
it is
       deleted. BindTexture may also be used to bind an existing texture object to either
       TEXTURE_2D, TEXTURE_CUBE_MAP, or TEXTURE_STREAM_IMG.
       ...
       In the initial state, TEXTURE_2D, TEXTURE_CUBE_MAP and TEXTURE_STREAM_IMG have two
       dimensional, cube map, and stream texture state vectors respectively associated with
them.
       In order that access to these initial textures not be lost, they are treated as
texture
       objects all of whose names are 0. The initial two-dimensional, cube map and stream
texture
       are therefore operated upon, queried, and applied as TEXTURE_2D, TEXTURE_CUBE_MAP or
       TEXTURE_STREAM_IMG respectively while 0 is bound to the corresponding targets.
       ...
       If a texture that is currently bound to one of the targets TEXTURE_2D,
TEXTURE_CUBE_MAP, or
```

```
       TEXTURE_STREAM_IMG is deleted, it is as though BindTexture had been executed with the
same
       target and texture zero.


   Modify Section 6.1.3, Enumerated Queries, p. 121
       (Modify the sentence after the GetTexParameter{if}v prototype p.121)
       ...
         The command
              void GetTexParameter{if}v( enum target, enum value, T data );
         returns information about target, which may be one of TEXTURE_2D, TEXTURE_CUBE_MAP,
or
       TEXTURE_STREAM_IMG, indicating the currently bound two-dimensional, cube map or
stream
       texture object.


Errors
       TBC

New Keywords

    samplerStreamIMG


Grammar changes

    The token SAMPLERSTREAMIMG is added to the list of tokens returned from lexical
    analysis and the type_specifier_no_prec production.

New Built-in Functions

    textureStreamIMG()

New Macro Definitions

    #define GL_IMG_texture_stream2 1

Additions to Chapter 3 of the OpenGL ES Shading Language specification:
       Add to Section 3.7, Keywords, p. 16

       samplerStreamIMG

Additions to Chapter 4 of the OpenGL ES Shading Language specification:

    Add the following to the table of basic types in section 4.1:

    Type:
        samplerStreamIMG

    Meaning:
        a handle for accessing a stream texture

   Add the following to the section beginning "The fragment language has the following
   predeclared globally scoped default precision statement" in section 4.5.3
              precision lowp samplerStreamIMG;

Additions to Chapter 8 of the OpenGL ES Shading Language specification:

    Add the following to the table of built-in functions in section 8.7:

    The built-in texture lookup functions textureStreamIMG and textureStreamProjIMG
    are optional, and must be enabled by

    #extension GL_IMG_texture_stream2 : enable

    before being used.

    Syntax:
        vec4 textureStreamIMG (samplerStreamIMG sampler, vec2 coord)
        vec4 textureStreamProjIMG (samplerStreamIMG sampler, vec3 coord)
        vec4 textureStreamProjIMG (samplerStreamIMG sampler, vec4 coord)

    Description:
        Use the texture coordinate coord to do a texture lookup in the Stream
        texture currently bound to sampler.  For the projective ("Proj")
```

```
        versions, the texture coordinate (coord.s, coord.t) is divided by the
        last component of coord. The third component of coord is ignored for
        the vec4 coord variant.

Additions to Chapter 9 of the OpenGL ES Shading Language specification:

   Add the following to the first paragraph of the shading language grammar in
   section 9 (p. 73):

              SAMPLERSTREAMIMG

   Add the following to the type_specifier_no_prec section of the shading language grammar
in
   section 9 (p. 79):

              SAMPLERSTREAMIMG

Errors

    TBC.

New State

Get Value                        Type    Get Command     Value    Description
---------                        ----    -----------     -----    -----------
TEXTURE_BINDING_STREAM_IMG        Z+     GetIntegerv       0      texture object
                                                                  bound to TEXTURE_STREAM_IMG

TEXTURE_NUM_STREAM_DEVICES_IMG    Z+     GetIntegerv       0      Number of stream devices
supported

TEXTURE_STREAM_DEVICE_WIDTH_IMG   Z+     GetTexStreamDeviceAttributeivIMG
                                                           0      Width of texture stream
device buffer

TEXTURE_STREAM_DEVICE_FORMAT_IMG Z2      GetTexStreamDeviceAttributeivIMG
                                                         GL_RGBA  Format of texture stream
device buffer

TEXTURE_STREAM_DEVICE_NUM_BUFFERS_IMG Z+ GetTexStreamDeviceAttributeivIMG
                                                           0      Number of texture stream
device buffers


Revision History

0.3 24/11/2008    Ben Bowman Further tidy up
0.2 24/10/2008    Ben Bowman Rewrote against full specification (2.0.23)
0.1 16/07/2008    Ben Bowman First revision
```

Appendix C. Supported Pixel Formats

Any Buffer Class device can export one of the following pixel formats:

RGB:

- Pixel Format: PVRSRV_PIXEL_FORMAT_RGB565
- Width: 0 to 2048
- Stride: ((Width * 2) + 31) & ~31
- Flags: n/a

RGBA:

- PVRSRV_PIXEL_FORMAT_ARGB8888:
- Width: 0 to 2048

Stride: ((Width * 4) + 31) & ~31

- PVRSRV_PIXEL_FORMAT_ARGB4444:
- Width: 0 to 2048

Stride: ((Width * 2) + 31) & ~31

Flags: n/a

Interleaved YUV 422:

PVRSRV_PIXEL_FORMAT_FOURCC_ORG_UYVY:

- 32 bit word encoding a pair of pixels: Y1 V Y0 U (LSB)
- Equivalent to http://www.fourcc.org/yuv.php#UYVY

PVRSRV_PIXEL_FORMAT_FOURCC_ORG_YUYV:

- 32 bit word encoding a pair of pixels: V Y1 U Y0 (LSB)
- Equivalent to http://www.fourcc.org/yuv.php#YUY2
- Width: 0 to 2048
- Stride: ((Width * 2) + 31) & ~31
- Flags:
    - PVRSRV_BC_FLAGS_YUVCSC_CONFORMANT_RANGE
      Inputs are in the range Y[16,235], UV[16, 239]
    - PVRSRV_BC_FLAGS_YUVCSC_FULL_RANGE
      Inputs are in the range Y[0,255], UV[0, 255]
    - PVRSRV_BC_FLAGS_YUVCSC_BT601
      YUV colour space conforms to ITU.BT-601
    - PVRSRV_BC_FLAGS_YUVCSC_BT709
      YUV colour space conforms to ITU.BT-709

Planar YUV 420 (NV12):

PVRSRV_PIXEL_FORMAT_NV12:

- Consists of an 8 bit per pixel Y plane followed immediately by an 8 bit 2x2 sub-sampled UV plane. In this case UV plane consists of 16 bit words encoding a 2x2 block of pixels: V U (LSB).
- Equivalent to http://www.fourcc.org/yuv.php#NV12
- Width: 0 to 2048
- Y Plane Stride: (Width + 31) & ~31
- UV Plane Stride: ((Width/2) + 31) & ~31
  Note: The buffer device stride is defined as the Y plane stride
- Flags:

- PVRSRV_BC_FLAGS_YUVCSC_CONFORMANT_RANGE
  Inputs are in the range Y[16,235], UV[16, 239]
- PVRSRV_BC_FLAGS_YUVCSC_FULL_RANGE
  Inputs are in the range Y[0,255], UV[0, 255]
- PVRSRV_BC_FLAGS_YUVCSC_BT601
  YUV colour space conforms to ITU.BT-601
- PVRSRV_BC_FLAGS_YUVCSC_BT709
  YUV colour space conforms to ITU.BT-709

For DDK version 1.6 and above:

3 Planar YUV I420 :

PVRSRV_PIXEL_FORMAT_I420:

- Consists of an 8 bit per pixel Y plane followed immediately by an 8 bit 2x2 sub-sampled U plane followed immediately by an 8 bit 2x2 sub-sampled V plane.
- Equivalent to http://www.fourcc.org/yuv.php#IYUV
- Width: 0 to 2048
- Y Plane Stride: (Width + 31) & ~31
- U / V Plane Stride: ((Width/2) + 31) & ~31
  Note: The buffer device stride is defined as the Y plane stride
- Flags:
  - PVRSRV_BC_FLAGS_YUVCSC_CONFORMANT_RANGE
    Inputs are in the range Y[16,235], UV[16, 239]
  - PVRSRV_BC_FLAGS_YUVCSC_FULL_RANGE
    Inputs are in the range Y[0,255], UV[0, 255]
  - PVRSRV_BC_FLAGS_YUVCSC_BT601
    YUV colour space conforms to ITU.BT-601
  - PVRSRV_BC_FLAGS_YUVCSC_BT709
    YUV colour space conforms to ITU.BT-709

Stream stride:

For SGX 530 1.1.1 and below and SGX 535 1.1.3 and below, the surface stride must be equal to the width rounded up to the nearest 32 pixels, as documented above.

For DDK version 1.5 and below:

On all other SGX cores the surface stride must be equal to the width rounded up to the nearest 8 pixels.

For DDK version 1.6 and above:

On all other SGX cores the surface stride can be unrelated to the width, although the stride must be a multiple of 16 bytes in this case.

YUV Format restriction:

Note that it is not possible to use buffer devices of different YUV formats within a single render. Due to the limitations of SGX HW in it's colour space conversion of YUV, only one set of CSC coefficients can be used per render.

Appendix D. Driver Dynamic Load and Registration

The loading of the 3rd party driver and Services registration can be done dynamically. Unload and the call to pfnPVRSRVRemoveBCDevice can also be done dynamically but the pfnPVRSRVRemoveBCDevice call will fail if client applications still have open connections to the BC driver device.