PVR2D

Software Functional Specification

Filename	:	PVR2D Driver.Software Functional Specification
Version	:	1.1.216 External Issue
Issue Date	:	09 Aug 2011
Author	:	PowerVR

1. Contents

Function API Listing

List of Figures

2. Introduction

This document describes how to use the PVR2D API used for controlling the 2D rendering by MBX and SGX hardware. PVR2D is used in the Linux and Symbian Operating Systems. It may also be applicable to other environments.

2.1. Architecture

The PVR2D library resides in user space and is compiled as a shared library to which applications link. The following diagram shows the interactivity between different components.

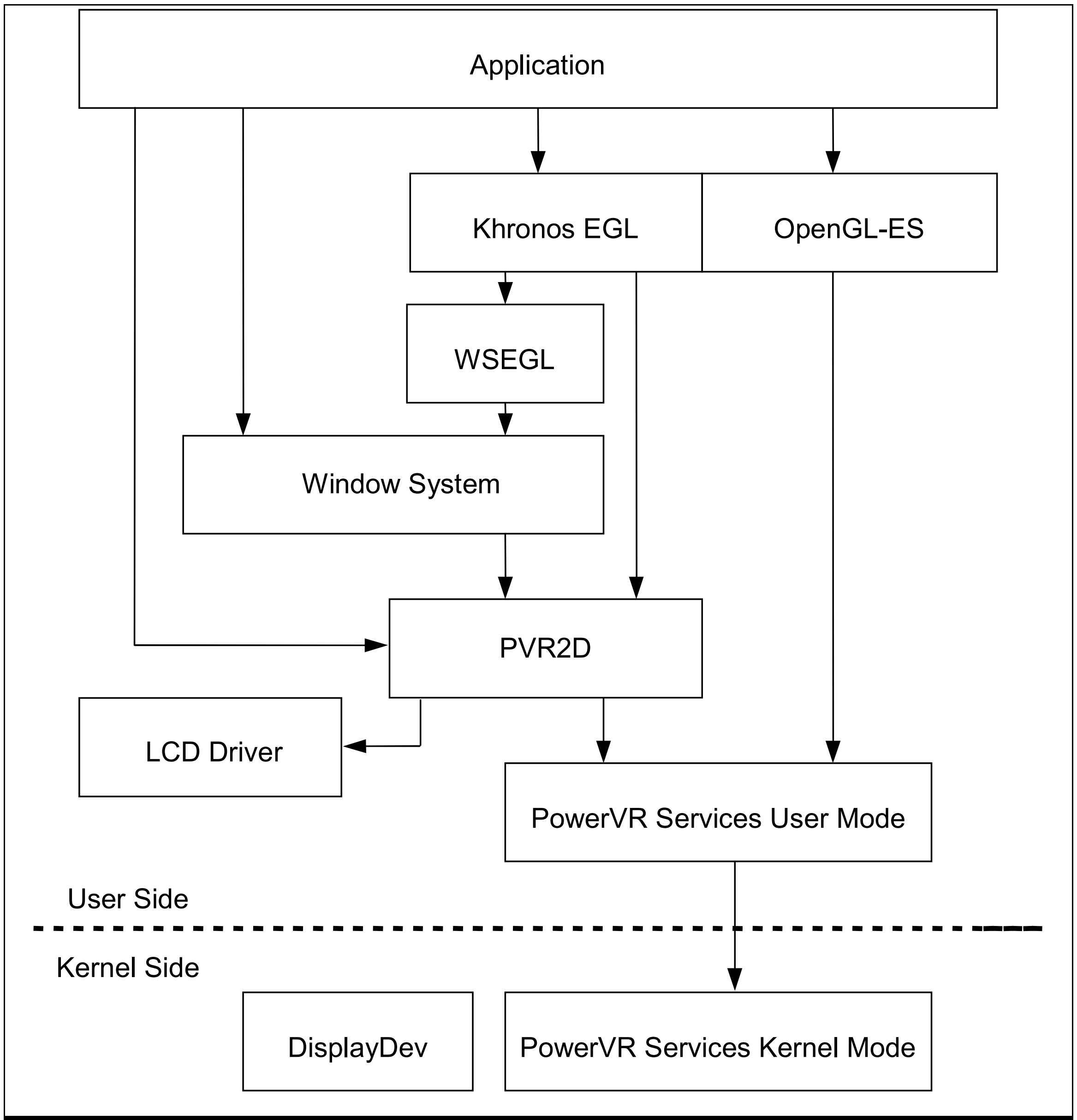


Figure 1 - PVR2D Architecture

2.2. Design Advantages

The library is in user space making it easier to maintain and debug.

The library is suitable for more than one operating system.

The frame buffer address and other memory addresses important to an application (like the Primary Surface) are stored in the PVR2DCONTEXT structure passed to the functions in the 2D API.

The PVR2D API does its own mapping of the memory allocated (which in most cases is the primary surface and subsequent flip surfaces) and it will expose the mapping into the PVR2DCONTEXT structure.

3. APIs

3.1. Functional Groups

The API calls can be grouped according to the following functionality:

- Device Enumeration
    - PVR2DEnumerateDevices
- Device Initialisation/De-initialisation
    - PVR2DCreateDeviceContext
    - PVR2DDestroyDeviceContext
    - PVR2DGetDeviceInfo
    - PVR2DGetMiscDisplayInfo
    - PVR2DGetFrameBuffer
    - PVR2DGetScreenMode
- Device Memory Management
    - PVR2DMemAlloc
    - PVR2DMemWrap
    - PVR2DMemExport
    - PVR2DMemMap
    - PVR2DMemFree
- Device Memory Synchronisation
    - PVR2DModifyPendingOps
    - PVR2DModifyCompleteOps
    - PVR2DFlushToSyncModObj
    - PVR2DTakeSyncToken
    - PVR2DReleaseSyncToken
    - PVR2DFlushToSyncToken
- Blit Functions
    - PVR2DBlt
    - PVR2DBlitClipped
    - PVR2DQueryBlitsComplete
- Present Blit Functions
    - PVR2DSetPresentBltProperties
    - PVR2DPresentBlt
- Present Flip Functions
    - PVR2DCreateFlipChain
    - PVR2DDestroyFlipChain
    - PVR2DGetFlipChainBuffers
    - PVR2DSetPresentFlipProperties
    - PVR2DPresentFlip
- 3D Blt Functions
    - PVR2DLoadUseCode
    - PVR2DBlt3D
    - PVR2DFreeUseCode
- Other functions

- PVR2DGetAPIRev
- PVR2DSet1555Alpha
- PVR2DWaitForNextHardwareEvent

3.2. Calling Order

A typical application or use of PVR2D will call the API's in the following order:

1. PVR2DEnumerateDevices
2. PVR2DCreateDeviceContext
3. Create the memory to be rendered to:
   - PVR2DMemAlloc
   - PVR2DMemWrap
4. Set up the screen to work with:
   - PVR2DGetScreenMode
   - PVR2DGetFrameBuffer
5. Perform the 2D operations needed:
   - PVR2DBlt
   - PVR2DQueryBlitsComplete
6. PVR2DMemFree
7. PVR2DDestroyDeviceContext

4. Assumptions

- Some blits (Colour Key, Alpha Blend, Pattern Copy) must be enabled using a special blit flag.
- The system surface & any flip chain buffers will be allocated by the display device in its memory. PVR2D is aware of the system surface and can return it, but does not own it.
- Unused structure parameters must be set to zero.

5. Differences between MBX and SGX versions of PVR2D

In general, the SGX version of PVR2D offers less functionality than the MBX version. This is due to differences between the respective 2D cores. A detailed list of differences follows:

Feature	MBX	SGX
Maximum number of clipping rectangles	4	1
Scaled blits	Supported	Not Supported
24 bit pixel formats	Supported (RGB888)	Mapped to 32 bits (ARGB0888)
Maximum size of pattern bitmap	31 by 31	16 by 16
Mask bitmap wraparound	Yes	No
Fill colour and colour key format	Destination surface format	ARGB8888

6. SGX Cores without the 2D HW

Some SGX cores do not have a 2D Core and so the PVR2D API is implemented using the 3D Core.

In order to allow efficient handling of 2D surfaces using the 3D core the surface stride should be set to a multiple of 32 pixels.

Implementation limitations with no 2D core are listed below:

Feature	SGX DDK 1.2	SGX DDK 1.3	SGX DDK 1.4 &1.5
1 bit per pixel masks	Not supported	Not supported	Not supported
1 and 4 bpp source	Not supported	Not supported	4 bpp Supported
8 bit Palletised source	Not supported	Not supported	Supported
Fully specified Alpha blend	Not supported	Not supported	Not supported
Unaligned stride	Not supported	Not supported	Not supported
Colour Key	Not supported	Not supported	Supported
Negative Stride	Not supported	Supported	Supported
PVR2DMemMap	Not supported	Supported	Supported
Patterns	Not supported	Not supported	Not supported
All other API Entry Points	Supported	Supported	Supported

Note that when the API is implemented using the 3D Core the combination of rotation with alpha blend or colour key or palletised source is not supported.

7. Common Constants and Enumerated Types

This section presents a list of common enumerated types and constants used throughout PVR2D API functions.

7.1. 2D Colour Bit Depths

Source and target colour bit depths can be specified for pixmaps, and primary surfaces using enumerated type PVR2DFORMAT.

PVR2DFORMAT	Description
`PVR2D_1BPP`	1 bit per pixel (used to create masks and for Font Expansion), also can be used as a 1 bit palletized source (2 colours).
`PVR2D_RGB565`	16 bits in RGB565 format (blue lowest 5 bits)
`PVR2D_ARGB4444`	16 bits in ARGB4444 format (blue lowest 4 bits)
`PVR2D_RGB888`	**MBX:** 24 bits in RGB888 format (blue lowest 8 bits) **SGX:** Not Supported (except with PTLA core)
`PVR2D_ARGB8888`	32 bits in ARGB8888 format (blue lowest 8 bits)
`PVR2D_ARGB1555`	16 bits in ARGB1555 format (blue lowest 5 bits) (**SGX** only)
`PVR2D_ALPHA8`	8 bit alpha-only source, RGB is a constant and taken from the Colour field of the PVR2DBLTINFO structure. (**SGX** only)
`PVR2D_ALPHA4`	4 bit alpha-only source, RGB from Colour field. (**SGX** only)
`PVR2D_PAL2`	2 bit palletized source (4 colours in pPalMemInfo field of the PVR2DBLTINFO structure) (**SGX** only)
`PVR2D_PAL4`	4 bit palletized source (16 colours in pPalMemInfo) (**SGX** only)
`PVR2D_PAL8`	8 bit palletized source (256 colours in pPalMemInfo) (**SGX** only)

The following is a list of WSEGL 3D colour bit depths compatible with the corresponding PVR2D pixel map colour depths above.

WSEGLPixelFormat	PVR2DFORMAT
`WSEGL_PIXELFORMAT_565`	`PVR2D_RGB565`
`WSEGL_PIXELFORMAT_4444`	`PVR2D_ARGB4444`
`WSEGL_PIXELFORMAT_8888`	`PVR2D_ARGB8888`

No other PVR2D formats can be used for 3D.operations.

Other Formats have been added to support SGX types which have a PTLA core (Presentation and Texture Load Accelerator). Emulation of the PTLA via the 3D core is not supported and functionality is limited to that of the PTLA hardware. The PTLA formats are as follows:

PVR2DFORMAT	Description
`PVR2D_U8`	monochrome unsigned 8 bit
`PVR2D_U88`	monochrome unsigned 16 bit
`PVR2D_S8`	monochrome signed 8 bit
`PVR2D_YUV422_YUYV`	YUV 422 low-high byte order Y0,U,Y1,V

PVR2DFORMAT	Description
PVR2D_YUV422_UYVY	YUV 422 low-high byte order U,Y0,V,Y1
PVR2D_YUV422_YVYU	YUV 422 low-high byte order Y0,V,Y1,U
PVR2D_YUV422_VYUY	YUV 422 low-high byte order V,Y0,U,Y1
PVR2D_YUV420_2PLANE	Planar YUV420 with 2 planes – luma Y, and chroma UV.
PVR2D_YUV420_3PLANE	Planar YUV420 with 3 planes – one each for Y U and V.
PVR2D_2101010ARGB	32 bit argb 2-10-10-10
PVR2D_888RSGSBS	Signed 24 bit RGB
PVR2D_16BPP_RAW	16 bit raw (no format conversion – just copy/rotate/twiddle)
PVR2D_32BPP_RAW	32 bit raw “
PVR2D_64BPP_RAW	64 bit raw “
PVR2D_128BPP_RAW	128 bit raw “

The PVR2D formats can be extended as described below.

PVR2D Format fields	#define	
Bits 0..15		PVR2D pixel format
Bits 16..23	PVR2D_FORMAT_LAYOUT_STRIDED	Standard 2D strided surface
	PVR2D_FORMAT_LAYOUT_TILED	Tiled surface (PTLA)
	PVR2D_FORMAT_LAYOUT_TWIDDLED	Twiddled surface (PTLA)
Bit 31	PVR2D_FORMAT_PVRSRV	Low level 3D format extension - for blts via the 3D core only. When this bit is set, bits 0..15 are taken as type PVRSRV_PIXEL_FORMAT (defined in servicesext.h)

Note: PVR2D_FORMAT_PVRSRV is a low level 3D format extension for blts going via the SGX 3D core, and only certain formats will work - those that are not supported by the Transfer Queue layer for the specific platform will return `PVR2DERROR_INVALID_PARAMETER`.

PVR2DEnumerateDevices

```
int PVR2DEnumerateDevices(
                PVR2DDEVICEINFO           *pDevInfo
               );
```

Inputs

`pDevInfo`	The device info array to fill in (or 0)

Outputs

`pDevInfo`	The device info array to fill in (or 0)

Returns

`nNumDevices`	Number of devices found
`PVR2D_OK`	Success
`PVR2DERROR_DEVICE_UNAVAILABLE`	Device cannot complete request
`PVR2DERROR_MEMORY_UNAVAILABLE`	Not sufficient memory to satisfy the request

Description

If PVR2DEnumerateDevices is called with a NULL pointer for pDevInfo, the number of display devices present is returned. If a failure occurs inside the function, then a PVR2DERROR value will be returned. These values are negative integers meaning that if a value of zero or less is returned, then no devices are available.

If pDevInfo is a valid pointer then an array of PVR2DDEVICEINFO structures, whose length is the number of display devices, will be filled in. Each pvr2d device corresponds to a physical display device. When calling PVR2DEnumerateDevices in this way then the integer returned should be interpreted as being of PVR2ERROR type.

The PVR2DDEVICEINFO Structure

```
typedef struct _PVR2DDEVICEINFO
{
        unsigned long ulDevID;
        char szDeviceName[PVR2D_MAX_DEVICE_NAME];
}PVR2DDEVICEINFO;
```

`ulDevID`	Unique device identifier
`szDeviceName`	The name of the display device

PVR2DCreateDeviceContext

```
PVR2DERROR PVR2DCreateDeviceContext(
              unsigned long          ulDevID,
              PVR2DCONTEXTHANDLE*    phContext,
              unsigned long          ulFlags);
```

Inputs

`ulDevID`	Device ID returned in PVR2DDEVICEINFO from PVR2DEnumerateDevices, or 0xffffffff.
`ulFlags`	No flags are defined at this time

Outputs

`phContext`	Receives the Context once created

Returns

`PVR2D_OK`	Success
`PVR2DERROR_DEVICE_UNAVAILABLE`	No MBX or SGX device found
`PVR2DERROR_IOCTL_ERROR`	Generic error code

Description

In order to use PVR2D, this function must be called to create a device context.

The function can be used in two ways. If a valid display device ID argument is passed, the full PVR2D API can be used (subject to the features supported by the display device).

If an invalid device ID argument is passed (such as 0xffffffff), PVR2D can still be used to allocate memory, wrap external memory, and perform blits and clipped blits. Any PVR2D function associated with a display device, including the presentation blit and flip part of the API, cannot be used.

A device ID of 0xffffffff is always treated as invalid.

This function does not actually change the state of the display device in those cases where a valid display device ID argument is passed.

PVR2DDestroyDeviceContext

```
PVR2DERROR PVR2DDestroyDeviceContext(
                PVR2DCONTEXTHANDLE hContext
               );
```

Inputs

`hContext`	The context to destroy

Outputs

Returns

`PVR2D_OK`	Success
`PVR2DERROR_INVALID_CONTEXT`	The supplied context is invalid

Description

PVR2DDestroyDeviceContext should be called once for each display device. This guarantees that all resources used by the context are freed.

PVR2DGetFrameBuffer

```
PVR2DERROR PVR2DGetFrameBuffer(
               PVR2DCONTEXTHANDLE          hContext,
               int                         nHeap,
               PVR2DMEMINFO                **ppsMemInfo);
```

Inputs

`hContext`	The context to operate on
`nHeap`	Must be set to `PVR2D_FB_PRIMARY_SURFACE`

Outputs

`ppsMemInfo`	Will receive the address of the primary surface

Returns

`PVR2D_OK`	Success
`PVR2DERROR_DEVICE_UNAVAILABLE`	Device cannot complete request
`PVR2DERROR_INVALID_CONTEXT`	The context is invalid

Description

Will return the meminfo representing the system surface for the given display device. This surface will not change when flipping – ie. It does not track the surface currently being displayed. This surface will always be the target of presentation blits.

PVR2DGetScreenMode

```
PVR2DERROR PVR2DGetScreenMode (
               PVR2DCONTEXTHANDLE        hContext,
               PVR2DFORMAT               *pFormat,
               long                      *plWidth,
               long                      *plHeight,
               long                      *plStride,
               int                       *piRefreshRate);
```

Inputs

`hContext`	The context to operate on

Outputs

`pFormat`	The format of the screen (see `PVR2DFORMAT` enumerated constants)
`plWidth`	The width of the screen
`plHeight`	Height of the screen
`plStride`	Stride of the screen – can be negative
`piRefreshRate`	Refresh rate of the screen (Hz). Currently unsupported

Returns

`PVR2D_OK`	Success
`PVR2DERROR_DEVICE_UNAVAILABLE`	Device cannot complete request
`PVR2DERROR_INVALID_CONTEXT`	The context is invalid

Description

This function retrieves information about the current mode of the display for this context.

PVR2DGetDeviceInfo

```
PVR2DERROR PVR2DGetDeviceInfo (
               PVR2DCONTEXTHANDLE        hContext,
               PVR2DDISPLAYINFO          *pDisplayInfo);
```

Inputs

`hContext`	The context to operate on

Outputs

`pDisplayInfo`	Information about the display for this context.

Returns

`PVR2D_OK`	Success
`PVR2DERROR_INVALID_CONTEXT`	The context is invalid

Description

This function retrieves information about the current type, capabilities and mode of the display for this context.

The DisplayInfo Structure

```
typedef struct _PVR2DDISPLAYINFO
{
	unsigned long ulMaxFlipChains;
	unsigned long ulMaxBuffersInChain;
	PVR2DFORMAT eFormat;
	unsigned long ulWidth;
	unsigned long ulHeight;
	long lStride;
	unsigned long ulMinFlipInterval;
	unsigned long ulMaxFlipInterval;
}PVR2DDISPLAYINFO;
```

`ulMaxFlipChains`	Maximum number of flip chains supported.
`ulMaxBuffersInChain`	The maximum number of buffers in a flip chain.
`eFormat`	The current format of the system surface (see `PVR2DFORMAT` enumerated constants)
`ulWidth`	The current width of the system surface
`ulHeight`	The current height of the system surface
`lStride`	The current stride of the system surface (may be negative)
`ulMinFlipInterval`	The minimum flip interval supported by the device
`ulMaxFlipInterval`	The maximum flip interval supported by the device

PVR2DGetMiscDisplayInfo

```
PVR2DERROR PVR2DGetMiscDisplayInfo (
                PVR2DCONTEXTHANDLE          hContext,
                PVR2DMISCDISPLAYINFO        *pMiscDisplayInfo);
```

Inputs

`hContext`	The context to operate on

Outputs

`pMiscDisplayInfo`	Miscellaneous information about the display.

Returns

`PVR2D_OK`	Success
`PVR2DERROR_INVALID_CONTEXT`	The context is invalid

Description

This function retrieves miscellaneous information about the display from the third party display driver for this context.

The PVR2DMISCDISPLAYINFO Structure

```
typedef struct  PVR2MISCDISPLAYINFO
{
        PVR2D_ULONG ulPhysicalWidthmm;
        PVR2D_ULONG ulPhysicalHeightmm;
        PVR2D_ULONG ulUnused[10];

}PVR2DMISCDISPLAYINFO;
```

`ulPhysicalWidthmm`	Physical width of the display in mm
`ulPhysicalHeightmm`	Physical height of the display in mm
`ulUnused`	Unused fields

PVR2DMemAlloc

```
PVR2DERROR PVR2DMemAlloc (
                PVR2DCONTEXTHANDLE          hContext,
                unsigned long               ulBytes,
                unsigned long               ulAlign,
                unsigned long               ulFlags,
                PVR2DMEMINFO                **ppsMemInfo);
```

Inputs

`hContext`	Device context.
`ulBytes`	Bytes to allocate
`ulAlign`	Required alignment of buffer in bytes
`ulFlags`	Flags to control the allocation

Outputs

`ppsMemInfo`	Receives a pointer to the returned meminfo structure.

Returns

`PVR2D_OK`	Success
`PVR2DERROR_INVALID_PARAMETER`	Invalid heap, or size is zero
`PVR2DERROR_INVALID_CONTEXT`	The supplied context is invalid
`PVR2DERROR_DEVICE_UNAVAILABLE`	The device is not in a state to be able to complete the operation
`PVR2DERROR_MEMORY_UNAVAILABLE`	Not sufficient memory to satisfy the request, or requested heap type is not supported

Description

Memory allocated by the PVR2DMemAlloc function must be freed using PVR2DMemFree.
The ulAlign variable allows you to specify an arbitrary power of 2 alignment for your allocated buffer.
There are no flags currently defined and so the ulFlags variable should be set to 0.

The MemInfo Structure

```
typedef struct _PVR2DMEMINFO
{
    void             *pBase;
    unsigned long    ui32MemSize;
    unsigned long    ui32DevAddr;
    unsigned long    ulFlags;
    void             *hPrivateData;
    void             *hPrivateMapData;
}PVR2DMEMINFO, *PPVR2DMEMINFO;
```

`pBase`	User mode base address of the allocated memory
`ui32MemSize`	Size of the allocation including alignment.
`ui32DevAddr`	Device address of memory
`ulFlags`	Internal description of MemInfo type, this must not be modified
`hPrivateData`	Internal data handle; this must not be modified.
`hPrivateMapData`	Used as input to PVR2DMemMap

PVR2DMemWrap

```
PVR2DERROR PVR2DMemWrap (
               PVR2DCONTEXTHANDLE          hContext,
               void                        *pMem,
               unsigned long               ulFlags,
               unsigned long               ulBytes,
               unsigned long               alPageAddress[],
               PVR2DMEMINFO                **ppsMemInfo);
```

Inputs

`hContext`	Device context.
`pMem`	The user mode CPU virtual address of the memory to wrap.
`ulFlags`	Flags to control the wrapping. Only PVR2D_WRAPFLAG_CONTIGUOUS is currently defined. If this flag is not set the memory to be wrapped is not contiguous.
`ulBytes`	The size of the memory to wrap.
`alPageAddress`	Optional. A pointer to an array of physical addresses of the pages to wrap. If the PVR2D_WRAPFLAG_CONTIGUOUS flag is set, only 1 page address containing the base of the memory to be wrapped needs to be passed in through this array. If the flag is not set, this array must include the addresses of all the pages the memory buffer spans.

Outputs

`ppsMemInfo`	Receives a pointer to the returned meminfo structure.

Returns

`PVR2D_OK`	Success
`PVR2DERROR_INVALID_PARAMETER`	Invalid number of pages or page list
`PVR2DERROR_INVALID_CONTEXT`	The supplied context is invalid
`PVR2DERROR_DEVICE_UNAVAILABLE`	The device is not in a state to be able to complete the operation
`PVR2DERROR_MEMORY_UNAVAILABLE`	Not sufficient memory to satisfy the request

Description

This function maps the already-allocated memory into the HW's memory space. The memory to be mapped can be contiguous or non-contiguous, according to the ulFlags parameter. If the memory is contiguous, only the page address containing the base of the allocation needs to be supplied, but if the memory is non-contiguous, a complete page address list must be specified. The offset within a page of the User mode CPU virtual address (pMem) will be used for the alignment of the device address mapping of the memory.

Memory wrapped by the PVR2DMemWrap function must have a corresponding PVR2DMemFree call before the underlying memory is freed.

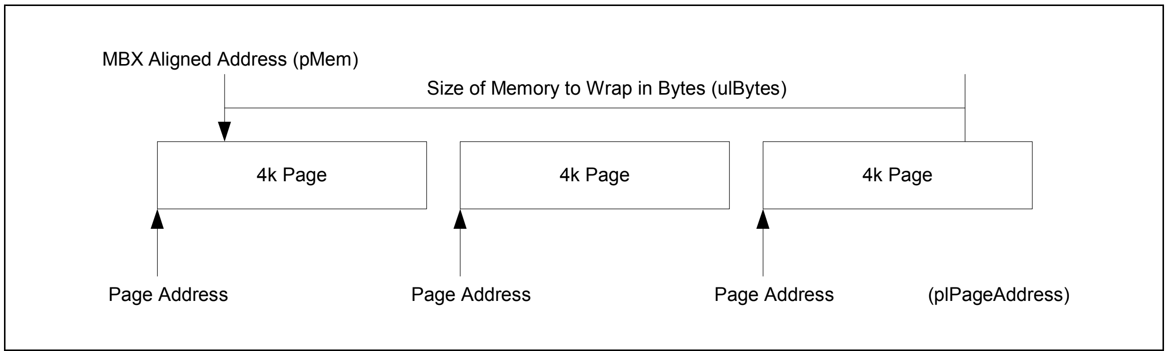


Figure 2 - Page Address List

PVR2DMemMap

```
PVR2DERROR PVR2DMemMap (
               PVR2DCONTEXTHANDLE        hContext,
               unsigned long             ulFlags,
               PVR2D_HANDLE              hMemHandle,
               PVR2DMEMINFO              **ppsDstMem);
```

Inputs

`hContext`	Device context.
`ulFlags`	Not used
`hMemHandle`	Handle from PVR2DMemExport

Outputs

`ppsDstInfo`	Receives a pointer to the returned meminfo structure.

Returns

`PVR2D_OK`	Success
`PVR2DERROR_INVALID_PARAMETER`	Invalid number of pages or page list
`PVR2DERROR_INVALID_CONTEXT`	The supplied context is invalid
`PVR2DERROR_DEVICE_UNAVAILABLE`	The device is not in a state to be able to complete the operation
`PVR2DERROR_MEMORY_UNAVAILABLE`	Not sufficient memory to satisfy the request

Description

PVR2DMemMap allows PVR2D memory that has been exported by another process to be mapped into the address space of the calling process (see PVR2DMemExport) If the owner of the memory has not exported it then PVR2DMemMap will fail. Only hMemHandle needs to be passed for the original buffer.

Memory mapped by the PVR2DMemMap function must have a corresponding PVR2DMemFree call in order to free the low level mapping structures and to relinquish the link to the exported memory.

NOTE: If the DDK is built with the build flag SUPPORT_SECURE_FD_EXPORT enabled (which it is by default) then the increased security will lock out PVR2DMemMap so it will fail. A basic choice between security or memory mapping functionality has to be made, however on certain platforms both can be achieved if the exporting process cooperates to give the mapping process access to the memory. For example, on one platform this is achieved by sending the file descriptor associated with the memory over a Unix socket which has the effect of making the file descriptor valid in the receiving process.

PVR2DMemExport

```
PVR2DERROR PVR2DMemExport (
               PVR2DCONTEXTHANDLE        hContext,
               unsigned long             ulFlags,
               PVR2DMEMINFO              *psMemInfo,
               PVR2D_HANDLE              *phMemHandle);
```

Inputs

`hContext`	Device context.
`ulFlags`	Not used
`psMemInfo`	PVR2D memory the calling process has allocated.

Outputs

`phMemHandle`	Receives the export handle that can be passed to other processes to allow access via PVR2DMemMap.

Returns

`PVR2D_OK`	Success
`PVR2DERROR_INVALID_PARAMETER`	Invalid number of pages or page list
`PVR2DERROR_INVALID_CONTEXT`	The supplied context is invalid
`PVR2DERROR_DEVICE_UNAVAILABLE`	The device is not in a state to be able to complete the operation

Description

PVR2DMemExport allows PVR2D memory to be exported so that other processes can gain access to the memory via the PVR2DMemMap call.

PVR2DMemFree

```
PVR2DERROR PVR2DMemFree (
                PVR2DCONTEXTHANDLE         hContext,
                PVR2DMEMINFO               *psMemInfo);
```

Inputs

`hContext`	The context to operate on
`psMemInfo`	Pointer to the MemInfo to free

Outputs

Returns

`PVR2D_OK`	Success
`PVR2DERROR_INVALID_CONTEXT`	Invalid hContext
`PVR2DERROR_INVALID_PARAMETER`	Invalid psMemInfo

Description

Memory allocated by the PVR2DMemAlloc function must be freed using PVR2DMemFree.

Memory mapped using PVR2DMemWrap must have a corresponding call to PVR2DMemFree before the underlying memory is freed.

Memory mapped by the PVR2DMemMap function must have a corresponding call to PVR2DMemFree in order to free the low level mapping structures and to relinquish the link to the exported memory.

PVR2DModifyPendingOps

```
PVR2DERROR PVR2DModifyPendingOps(
               const PVR2DCONTEXTHANDLE hContext,
               PVR2D_HANDLE *phSyncModObj,
               PVR2DMEMINFO *psMemInfo,
               PVR2D_BOOL  bIsWriteOp,
               PVR2D_ULONG *pulReadOpsPending,
               PVR2D_ULONG *pulWriteOpsPending);
```

Inputs

`hContext`	The context to operate on
`psMemInfo`	Pointer to the MemInfo to operate on
`bIsWriteOp`	Flag denoting modification target (reads or writes)

Outputs

`phSyncModObj`	Pointer to a handle to receive the tracking object for this modification
`pulReadOpsPending`	(optional) Pointer to variable to receive the number of read ops pending at the point this modification was made.
`pulWriteOpsPending`	(optional) Pointer to variable to receive the number of write ops pending at the point this modification was made.

Returns

`PVR2D_OK`	Success
`PVR2DERROR_INVALID_CONTEXT`	Invalid `hContext`
`PVR2DERROR_INVALID_PARAMETER`	Invalid `psMemInfo` or `phSyncModObj`
`PVR2DERROR_GENERIC_ERROR`	The modification can not be made

Description

This API is used to flag an outstanding read or write operation to the passed in Mem Info. This is done by incrementing a pending operations counter (read or write) associated with the memory. The API returns a modification object handle by which the user can refer to the modification. The user must call PVR2DFlushToSyncModObj using the supplied object handle to wait for all previous operations to this surface to be complete. At this point the user can perform their read\write to the surface. The user must then call PVR2DModifyCompleteOps using the supplied handle to mark the operation as complete. This will allow the surface to be accessed\modified by other entities in the system that were waiting for the operation to complete. **Note that every call to PVR2DModifyPendingOps MUST have corresponding calls, first to PVR2DFlushToSyncModObj, then finally to PVR2DModifyCompleteOps. Failure to honour this sequence will stall the graphics pipeline permanently**.

PVR2DModifyCompleteOps

```
PVR2DERROR PVR2DModifyCompleteOps(
               const PVR2DCONTEXTHANDLE hContext,
               PVR2D_HANDLE    hSyncModObj);
```

Inputs

`hContext`	The context to operate on
`hSyncModObj`	Handle to the modification object to complete

Outputs

Returns

`PVR2D_OK`	Success
`PVR2DERROR_INVALID_CONTEXT`	Invalid `hContext`
`PVR2DERROR_INVALID_PARAMETER`	Invalid `hSyncModObj`
`PVR2DERROR_GENERIC_ERROR`	The modification can not be made

Description

This API is used to mark an outstanding read or write operation created with PVR2DModifyPendingOps as complete. This is done by incrementing a complete operations counter (read or write) associated with the memory. The user must call PVR2DFlushToSyncModObj using the supplied object handle to wait for all previous operations to this surface to be complete before calling this API. **Note that every call to PVR2DModifyPendingOps MUST have corresponding calls, first to PVR2DFlushToSyncModObj, then finally to PVR2DModifyCompleteOps. Failure to honour this sequence will stall the graphics pipeline permanently**.

PVR2DFlushToSyncModObj

```
PVR2DERROR PVR2DFlushToSyncModObj(
               const PVR2DCONTEXTHANDLE hContext,
               PVR2D_HANDLE hSyncModObj,
               PVR2D_BOOL bWait);
```

Inputs

`hContext`	The context to operate on
`hSyncModObj`	Handle to the modification object to flushed to
`bWait`	Flag denoting whether API should block until operations are flushed

Outputs

Returns

`PVR2D_OK`	All preceding operations are complete.
`PVR2DERROR_INVALID_CONTEXT`	Invalid `hContext`
`PVR2DERROR_INVALID_PARAMETER`	Invalid `hSyncModObj`
`PVR2DERROR_GENERIC_ERROR`	The flush can not be performed
`PVR2DERROR_BLT_NOTCOMPLETE`	The preceding operations are not yet complete

Description

This API is used to flush all read and write operations that precede the operation tracked by. `hSyncModObj`. The API can block until these operations are complete by passing the `bWait` flag as `PVR2D_TRUE`. If the wait is not requested the API will perform a single test and return. Once all preceding operations are complete the user is free to perform their read or write from\to the associated memory. **Note that every call to PVR2DModifyPendingOps MUST have corresponding calls, first to PVR2DFlushToSyncModObj, then finally to PVR2DModifyCompleteOps. Failure to honour this sequence will stall the graphics pipeline permanently**.

PVR2DTakeSyncToken

```
PVR2DERROR PVR2DTakeSyncToken(
               const PVR2DCONTEXTHANDLE hContext,
               PVR2DMEMINFO *psMemInfo,
               PVR2D_HANDLE *phSyncToken,
               PVR2D_ULONG *pulReadOpsPending,
               PVR2D_ULONG *pulWriteOpsPending);
```

Inputs

`hContext`	The context to operate on
`psMemInfo`	Pointer to the MemInfo to operate on

Outputs

`phSyncToken`	Pointer to a handle to receive the snapshot tracking token.
`pulReadOpsPending`	(optional) Pointer to variable to receive the number of read ops pending at the point this snapshot was taken.
`pulWriteOpsPending`	(optional) Pointer to variable to receive the number of write ops pending at the point this snapshot was taken.

Returns

`PVR2D_OK`	Success
`PVR2DERROR_INVALID_CONTEXT`	Invalid `hContext`
`PVR2DERROR_INVALID_PARAMETER`	Invalid `psMemInfo` or `phSyncToken`
`PVR2DERROR_MEMORY_UNAVAILABLE`	Cannot allocate resources for the token.

Description

This API is used to snapshot a particular place in the operation sequence for a given piece of memory. A token is returned that references the relevant information to allow the user to wait for all operations preceding the token to be complete. This is achieved by a call to PVR2DFlushToSyncToken. Note that the API does not modify the synchronisation data in any way. Once the user has finished with the token they must call PVR2DReleaseSyncToken to release the associated resources.

PVR2DReleaseSyncToken

```
PVR2DERROR PVR2DReleaseSyncToken(
               const PVR2DCONTEXTHANDLE hContext,
               PVR2D_HANDLE hSyncToken);
```

Inputs

`hContext`	The context to operate on
`hSyncToken`	Handle to the snapshot token to release

Outputs

Returns

`PVR2D_OK`	Success
`PVR2DERROR_INVALID_CONTEXT`	Invalid `hContext`
`PVR2DERROR_INVALID_PARAMETER`	Invalid `hSyncToken`

Description

This API is used to free the resources associates with a snapshot token previously created with PVR2DTakeSyncToken.

PVR2DFlushToSyncToken

```
PVR2DERROR PVR2DFlushToSyncToken(
               const PVR2DCONTEXTHANDLE hContext,
               PVR2DMEMINFO *psMemInfo,
               PVR2D_HANDLE hSyncToken,
               PVR2D_BOOL bWait);
```

Inputs

`hContext`	The context to operate on
`psMemInfo`	Pointer to the MemInfo to operate on
`hSyncToken`	Handle to the snapshot token to flushed to
`bWait`	Flag denoting whether API should block until operations are flushed

Outputs

Returns

`PVR2D_OK`	All preceding operations are complete.
`PVR2DERROR_INVALID_CONTEXT`	Invalid `hContext`
`PVR2DERROR_INVALID_PARAMETER`	Invalid `hSyncToken` or `psMemInfo`
`PVR2DERROR_GENERIC_ERROR`	The flush can not be performed
`PVR2DERROR_BLT_NOTCOMPLETE`	The preceding operations are not yet complete

Description

This API is used to flush all read and write operations that precede the snapshot token. `hSyncToken`. The API can block until these operations are complete by passing the `bWait` flag as `PVR2D_TRUE`. If the wait is not requested the API will perform a single test and return. The API can be used to wait for a particular place in the operation sequence timeline of a MemInfo. **It is not** designed to allow the user to read or write access to the surface. Once a token has been flushed it must be released via a call to PVR2DReleaseSyncToken.

PVR2DBlt

```
int PVR2DBlt(
               PVR2DCONTEXTHANDLE         hContext,
               PVR2DBLTINFO               *pBltInfo);
```

Inputs

`hContext`	The context to operate on
`pBltInfo`	Structure describing the blit

Outputs

`None`

Returns

`PVR2D_OK`	Success
`PVR2DERROR_DEVICE_UNAVAILABLE`	Device cannot complete request
`PVR2DERROR_INVALID_CONTEXT`	The context is invalid

Description

Perform a blit operation.

It should be assumed that the issued blit operation has not completed when this function returns. If synchronisation with the blit is required, the PVR2DQueryBlitsClomplete function should be called to determine if all currently submitted blits have completed on the destination surface.

See Appendix A for an illustration of the rectangles that are used in this function.

Please ensure that the Source Rectangle does not go outside the Source Surface and the Destination Rectangle does not go outside the Destination Surface. Also note that negative coordinates are not supported. Drawing outside a surface allocation may produce a page fault, in which case the blt will be aborted and the sync object will not complete. See Appendix A for more details. Note however that there is a build option that corrects any rectangle that goes outside its surface by calculating the intersection; the build option is `PVR2D_VALIDATE_INPUT_PARAMS`. When this is enabled there is a difference between debug and release builds; debug builds will return an error to highlight the incorrect parameters, and the release build will modify the input params to ensure that there is no drawing outside the surface. Without this build option no rectangle checking is done and it is assumed that the caller has already done this and does not want it repeated.

The BlitInfo structure is described below:

```
typedef struct _PVR2DBLTINFO
{
        unsigned long         CopyCode;
        unsigned long         Colour;
        unsigned long         ColourKey;
        unsigned long         ColourKeyMask;
        unsigned char         GlobalAlphaValue;
        unsigned char         AlphaBlendingFunc;
        PVR2DBLITFLAGS        BlitFlags;
        PVR2DMEMINFO          *pDstMemInfo;
        unsigned long         DstOffset;
        long                  DstStride;
        long                  DstX, DstY;
        long                  DSizeX,DSizeY;
        PVR2DFORMAT           DstFormat;
        unsigned long         DstSurfWidth;
        unsigned long         DstSurfHeight;
        PVR2DMEMINFO          *pSrcMemInfo;
        unsigned long         SrcOffset;
        long                  SrcStride;
        long                  SrcX, SrcY;
        long                  SizeX,SizeY;
        PVR2DFORMAT           SrcFormat;
        unsigned long         SrcSurfWidth;
        unsigned long         SrcSurfHeight;
        PVR2DMEMINFO          *pPalMemInfo;
        unsigned long         PalOffset;
        PVR2DMEMINFO          *pMaskMemInfo;
        unsigned long         MaskOffset;
        long                  MaskStride;
        long                  MaskX, MaskY;
        unsigned long         MaskSurfWidth;
        unsigned long         MaskSurfHeight;
        PPVR2D_ALPHABLT       pAlpha;
        PVR2D_ULONG           uSrcChromaPlane1;
        PVR2D_ULONG           uSrcChromaPlane2;
        PVR2D_ULONG           uDstChromaPlane1;
        PVR2D_ULONG           uDstChromaPlane2;
        PVR2D_ULONG           ColourKeyMask;
        PPVR2D_SURFACE        pPat;
        PVR2D_LONG            PatX, PatY;

}PVR2DBLTINFO, *PPVR2DBLTINFO;
```

`copyCode`	The Raster Operation to perform (rop code)
`Color`	The colour to fill with when the copyCode specifies colour fill
`ColorKey`	The colour key to use when BlitFlags specifies a color-keyed operation
`GlobalAlphaValue`	The 0-255 value specifying global alpha
`AlphaBlendingFunc`	The blending function used to specify the alpha-blending operations of the source and destination surfaces. The allowed functions are described in the table below.

`BlitFlags`	Flag used to enable additional blits like Colour Key, Global Alpha Blending, Alpha Channel (Per-Pixel) Blending, and Pattern Surface blit. The respective blits are enabled with the following flag constants:
	`PVR2D_BLIT_CK_ENABLE,`
	`PVR2D_BLIT_GLOBAL_ALPHA_ENABLE,`
	`PVR2D_BLIT_PERPIXEL_ALPHABLEND_ENABLE,`
	`PVR2D_BLIT_PAT_SURFACE_ENABLE`
	User defined alpha blend equation (SGX only):

	`PVR2D_BLIT_FULLY_SPECIFIED_ALPHA_ENABLE`
	Destination surface rotation (SGX only) :
	`PVR2D_BLIT_ROT_90`
	`PVR2D_BLIT_ROT_180`
	`PVR2D_BLIT_ROT_270`
	The overlapped blt copy order can be forced (SGX only):
	`PVR2D_BLIT_COPYORDER_TL2BR`
	`PVR2D_BLIT_COPYORDER_BR2TL`
	`PVR2D_BLIT_COPYORDER_TR2BL`
	`PVR2D_BLIT_COPYORDER_BL2TR`
	The colour key type can be specified (SGX only):
	`PVR2D_BLIT_COLKEY_SOURCE`
	`PVR2D_BLIT_COLKEY_DEST`
	`PVR2D_BLIT_COLKEY_MASKED`
	`PVR2D_BLIT_COLKEY_OP_PASS`
	`PVR2D_BLIT_COLKEY_OP_REJECT`
	The blt may be sent to 3D or 2D cores:
	`PVR2D_BLIT_PATH_2DCORE`
	`PVR2D_BLIT_PATH_3DCORE`
	The blt may be sent to the software blit engine:
	`PVR2D_BLIT_PATH_SWBLT`
	Do blt immediately without waiting for the source sync requirements:
	`PVR2D_BLIT_NO_SRC_SYNC_INFO`
	To write the value of the source SyncInfo's ReadOpsPending value to it's ReadOpsComplete member on completion of the blit:
	`PVR2D_BLIT_ISSUE_STATUS_UPDATES`
	Reserved for future use :
	`PVR2D_BLIT_ROP4`

`pDstMemInfo & DstOffset`	Pair to describe destination bitmap for a blit. See pSrcMemInfo for description. Always required
`DstStride`	Destination pixmap stride
`DstX & DstY`	The x,y position to blit to
`DSizeX & DSizeY`	The size of the destination rectangle
`DstFormat`	The pixel format of the destination pixmap
`DstSurfWidth & DstSurfHeight`	Destination surface size in pixels
`pSrcMemInfo & SrcOffset`	Pair to describe the source bitmap for a blit. pSrcMemInfo describes the base of the memory, srcOffset indicates the offset of the pixmap within that block of memory, required only when copyCode requires a source pixmap (is not a fill). Patterns surfaces are placed here for Pattern blits, and the corresponding pattern Blit Flag must be enabled (see BlitFlags). Initialize to NULL when not needed.
`SrcStride`	The stride of the source/pattern pixmap
`SrcX & SrcY`	The x,y offset from the start of the surface to the source rectangle. For patterns the offset from the start of the source surface to the source rectangle is always zero and SrcX and SrcY define the offset to the first pixel that is read, which must be < the pattern size.
`SizeX & SizeY`	The size of the source/pattern blt rectangle.
`SrcFormat`	The pixel format of the source/pattern pixmap
`pPalMemInfo & PalOffset`	Pair to describe a palette used to contain an argb8888 colour table used for palletised source or pattern surfaces (SGX only). The palette address, ie the base plus offset, must be on a 16 byte

	boundary.
`SrcSurfWidth & SrcSurfHeight`	Source surface size in pixels. For patterns this represents the pattern size (maximum size is 16x16 pixels for patterns).
`pMaskMemInfo & MaskOffset`	For masked copyCodes, describes the mask to apply to the blit. Mask is 1 bit deep (PVR2D_1BPP).
`MaskStride`	Signed stride of the mask surface
`MaskX, MaskY`	Mask blt rect, top left
`MaskSurfWidth & MaskSurfHeight`	Mask surface size in pixels
`pAlpha`	User defined alpha blend equation. See below for more details.
`uSrcChromaPlane1`	Offset from start of allocation to chroma plane 1 for YUV420 formats
`uSrcChromaPlane2`	Offset from start of allocation to chroma plane 2 for YUV420 formats
`uDstChromaPlane1`	Offset from start of allocation to chroma plane 1 for YUV420 formats
`uDstChromaPlane2`	Offset from start of allocation to chroma plane 2 for YUV420 formats
`ColourKeyMask`	32 bit colour key mask, only valid when PVR2D_BLIT_COLKEY_MASKED is set
`pPat, PatX, PatY`	Reserved for future use

Format modifiers : format fields such as `DstFormat` can have optional bits set as shown.

`PVR2D_FORMAT_LAYOUT_TILED`	The pixels are tiled
`PVR2D_FORMAT_LAYOUT_TWIDDLED`	The pixels are twiddled
`PVR2D_SURFACE_PDUMP`	Capture the surface to pdump file (only needed after host writes)
`PVR2D_BLTRECT_PDUMP`	Capture only the blt rectangle not the entire surface

The CopyCode parameter is a ropcode describing the operation to perform, but may be filled in using the easier to understand predefined copycode constants:

Common ROP Codes	**Description**
`PVR2DROPclear`	Blackness (all 0 values)
`PVR2DROPset`	Whiteness (all 1 values)
`PVR2DROPnoop`	Destination
Source and Destination ROP Codes	**Description**
`PVR2DROPand`	src AND dst
`PVR2DROPandReverse`	src AND NOT dst
`PVR2DROPcopy`	Src (Used for source copy, colour key and alpha blts)
`PVR2DROPandInverted`	NOT src AND dst
`PVR2DROPxor`	src XOR dst
`PVR2DROPor`	src OR dst
`PVR2DROPnor`	NOT src AND NOT dst

Common ROP Codes	Description
`PVR2DROPequiv`	NOT src XOR dst
`PVR2DROPinvert`	NOT dst
`PVR2DROPorReverse`	src OR NOT dst
`PVR2DROPcopyInverted`	NOT src
`PVR2DROPorInverted`	NOT src OR dst
`PVR2DROPnand`	NOT src OR NOT dst
Pattern ROP Codes	**Description**
`PVR2DPATROPand`	pat AND dst
`PVR2DPATROPandReverse`	pat AND NOT dst
`PVR2DPATROPcopy`	pat (Used for color fills and pattern blts)
`PVR2DPATROPandInverted`	NOT pat AND dst
`PVR2DPATROPxor`	pat XOR dst
`PVR2DPATROPor`	pat OR dst
`PVR2DPATROPnor`	NOT pat AND NOT dst
`PVR2DPATROPequiv`	NOT pat XOR dst
`PVR2DPATROPinvert`	NOT dst
`PVR2DPATROPorReverse`	pat OR NOT dst
`PVR2DPATROPcopyInverted`	NOT pat
`PVR2DPATROPorInverted`	NOT pat OR dst
`PVR2DPATROPnand`	NOT src OR NOT dst

A 16 bit ROP4 code is needed to define a mask blt, and is defined by two ROP3 codes, one each for the 1 and 0 state of the 1BPP mask surface. The CopyCode field of the PVR2DBLTINFO structure is taken to be an 8 bit ROP3 code when pMaskMemInfo is zero, otherwise it is assumed to be a 16 bit ROP4 and the mask is enabled. The PVR2DMASKROP4 macro can be used to construct a ROP4 from two ROP3's, where ROP3a is the ROP used when the mask pixel = 1, and ROP3b when the mask pixel = 0. A common application is a ROP4 of 0xAAF0 which is used for text glyphs, and this has been predefined in the PVR2DROP4MaskedFill macro :

```
#define PVR2DROP4MaskedCopy                 PVR2DROP4(PVR2DROPnoop,PVR2DROPcopy)
#define PVR2DROP4MaskedFill                 PVR2DROP4(PVR2DROPnoop,PVR2DPATROPcopy)
```

Alpha Blending

There are three types of Alpha blend :

a) Standard PVR2D Alpha Blend
b) Fully Specified Alpha Blend (SGX 2D Core only)
c) Alpha blend via 3D Core with custom pixel shader code (SGX only via PVR2DBlt3D API)

Standard PVR2D Alpha Blend

Standard global alpha is specified by setting the PVR2D_BLIT_GLOBAL_ALPHA_ENABLE flag and setting the GlobalAlphaValue field of PVR2DBLTINFO structure with the 8 bit alpha value, and gives the following result :

```
Cdst = Csrc*Aglob + Cdst*(1-Aglob)
```

Standard source alpha is specified by setting the PVR2D_BLIT_PERPIXEL_ALPHABLEND_ENABLE flag and setting the AlphaBlendingFunc field of the PVR2DBLTINFO structure as shown :

Alpha Blending Function	Description
`PVR2D_ALPHA_OP_SRC_DSTINV`	Linear source alpha blend: `Cdst = Csrc*Asrc + Cdst*(1-Asrc)`
`PVR2D_ALPHA_OP_SRCP_DSTINV`	Premultiplied source alpha blend : `Cdst = Csrc + Cdst*(1-Asrc)`

Standard Global + Source Alpha is defined by enabling both source and global alpha at the same time. This is only available for premultiplied sources, and gives the following result :

```
Cdest = Aglob*Csrc + (1 - (Aglob*Asrc))*Cdest
```

Fully Specified Alpha Blend (SGX 2D Core only)

A user defined alpha blend equation for the 2D Core can be defined by enabling the PVR2D_BLIT_FULLY_SPECIFIED_ALPHA_ENABLE flag. When this flag is set the pAlpha field of the PVR2DBLTINFO structure must point to a valid PVR2D_ALPHABLT structure.

A fully specified user defined Alpha Blend operation is defined as :

```
Adst = (ALPHA_1 * Asrc) + (ALPHA_3 * Adst)
Cdst = (ALPHA_2 * Csrc) + (ALPHA_4 * Cdst)
```

where ALPHA_1, ALPHA_2, ALPHA_3, and ALPHA_4 are the PVR2D_BLEND_OP factors defined in the PVR2D_ALPHABLT structure.

If bPremulAlpha is enabled then the equations become the following :

```
PRE_MUL = Asrc * Aglob
Adst = (ALPHA_1 * Asrc) + (PRE_MUL * Adst)
Cdst = (ALPHA_2 * Csrc) + (PRE_MUL * Cdst)
```

The PVR2D_ALPHABLT structure is shown below :

```
typedef enum
{
        PVR2D_BLEND_OP_ZERO = 0,
        PVR2D_BLEND_OP_ONE = 1,
        PVR2D_BLEND_OP_SRC = 2,
        PVR2D_BLEND_OP_DST = 3,
        PVR2D_BLEND_OP_GLOBAL = 4,
        PVR2D_BLEND_OP_SRC_PLUS_GLOBAL = 5,
        PVR2D_BLEND_OP_DST_PLUS_GLOBAL = 6
}PVR2D_BLEND_OP;

typedef struct _PVR2DBLTINFO
{
        PVR2D_BLEND_OP          eAlpha1;
        PVR2D_BOOL              bAlpha1Invert;
        PVR2D_BLEND_OP          eAlpha2;
        PVR2D_BOOL              bAlpha2Invert;
        PVR2D_BLEND_OP          eAlpha3;
        PVR2D_BOOL              bAlpha3Invert;
        PVR2D_BLEND_OP          eAlpha4;
        PVR2D_BOOL              bAlpha4Invert;
        PVR2D_BOOL              bPremulAlpha;
        PVR2D_BOOL              bTransAlpha;
        unsigned char           uGlobalRGB;
        unsigned char           uGlobalA;

} PVR2D_ALPHABLT, *PPVR2D_ALPHABLT;
```

Separate global alpha values are provided for RGB and Alpha channels and are specified in uGlobalRGB and uGlobalAlpha, and these must be specified if a blend OP with global alpha is used.

If bTransAlpha is enabled then a source pixel alpha of zero forces the source to be transparent for that pixel regardless of the blend equation being used.

MBX notes

No format conversion is applied, so fill colours and colour keys, as well as the pixel format of the pattern, source and mask bitmaps, must match the destination pixel format. No stretch is applied for pattern and mask bitmaps. Pattern and mask bitmaps will wraparound if they don't match the size of the destination. Source bitmaps will be scaled to match the size of the destination.

SGX Notes

Fill colours and colour keys must be in ARGB8888 format. An appropriate colour key bit mask must be defined for non 8888 formats, eg./ CKEY_MASK_565. Source and mask bitmaps must be at least the same size as the destination blt rectangle. Pattern bitmaps wrap and so they don't need to match the size of the destination.

PVR2DBltClipped

```
PVR2DERROR PVR2DBltClipped(
               PVR2DCONTEXTHANDLE        hContext,
               PVR2DBLTINFO              *pBltInfo,
               unsigned long             ulNumClipRects,
               PVR2DRECT                 *pClipRects);
```

Inputs

`hContext`	The context to operate on
`pBltInfo`	Structure describing the blit
`ulNumClipRects`	Number of clip rectangles
`pClipRects`	Clip rectangle list

Outputs

Returns

`PVR2D_OK`	Success
`PVR2DERROR_DEVICE_UNAVAILABLE`	Device cannot complete request
`PVR2DERROR_INVALID_CONTEXT`	The context is invalid

Description

Same semantics as blit.

MBX notes

Up to 4 clip rectangles can be specified.

SGX Notes

Only 1 clip rectangle can be specified.

PVR2DQueryBlitsComplete

```
PVR2DERROR PVR2DQueryBlitsComplete(
               PVR2DCONTEXTHANDLE         hContext,
               PVR2DMEMINFO               *pMemInfo,
               unsigned int               uiWaitForComplete);
```

Inputs

`hContext`	The context to operate on
`pMemInfo`	The PVR2D surface to wait for blits to complete from/to.
`uiWaitForComplete`	A flag to determine if the function should wait for blits to complete or should simply return the status of the pending blits immediately

Outputs

Returns

`PVR2D_OK`	Success
`PVR2DERROR_BLT_NOTCOMPLETE`	Blits are incomplete on this surface
`PVR2DERROR_DEVICE_UNAVAILABLE`	Device cannot complete request
`PVR2DERROR_INVALID_CONTEXT`	The context is invalid

Description

Tells the caller if all currently submitted blits are complete for a surface. The flag uiWaitForComplete allows the caller to specify whether to wait for blits to complete or simply return the status immediately.

SGX Notes

This function actually determines whether all operations on a surface have completed, not just blits.

PVR2DSetPresentBlitProperties

```
PVR2DERROR PVR2DSetPresentBlitProperties (
               PVR2DCONTEXTHANDLE hContext,
               unsigned long ulPropertyMask,
               long lSrcStride,
               unsigned long ulDstWidth,
               unsigned long ulDstHeight,
               long lDstXPos,
               long lDstYPos,
               unsigned long ulNumClipRects,
               PVR2DRECT *pClipRects,
               unsigned long ulSwapInterval);
```

Inputs

`hContext`	The context to operate on
`ulPropertyMask`	A mask of properties to set. Any of PVR2D_PRESENT_PROPERTY_SRCSTRIDE PVR2D_PRESENT_PROPERTY_DSTSIZE PVR2D_PRESENT_PROPERTY_DSTPOS PVR2D_PRESENT_PROPERTY_CLIPRECTS PVR2D_PRESENT_PROPERTY_INTERVAL may be Or'd in.
`lSrcStride`	The stride in bytes of the back buffer surface to be presented.
`ulDstWidth`	The width of the blit.
`ulDstHeight`	The height of the blit.
`lDstXPos`	The X position for the blit relative to the system surface origin
`lDstYPos`	The Y position for the blit relative to the system surface origin.
`ulNumClipRects`	The number of clip rectangles specified in the following parameter
`pClipRects`	A pointer to a list of rectangles describing the un-occluded areas of the surface to be presented.
`ulSwapInterval`	The number of display update periods to wait before a present takes place. Currently unsupported for blitting

Outputs

Returns

`PVR2D_OK`	Success
`PVR2DERROR_INVALID_PARAMETER`	A parameter is invalid
`PVR2DERROR_INVALID_CONTEXT`	The context is invalid

Description

Sets the properties for a presentation blt. The ulPropertyMask parameter allows any combination of the present properties to be set at once, by OR'ing the relevant bit into the mask and then supplying the parameter.

PVR2DPresentBlt

```
PVR2DERROR PVR2DPresentBlt (
               PVR2DCONTEXTHANDLE hContext,
               PVR2DMEMINFO *psMemInfo
               long lRenderID);
```

Inputs

`hContext`	The context to operate on
`psMemInfo`	The buffer to blt from.
`lRenderID`	Private data associated with this presentation blt. Currently unsupported.

Outputs

Returns

`PVR2D_OK`	Success
`PVR2DERROR_INVALID_PARAMETER`	A parameter is invalid
`PVR2DERROR_DEVICE_UNAVAILABLE`	Device cannot complete request
`PVR2DERROR_INVALID_CONTEXT`	The context is invalid

Description

The present blt properties are used to perform a (potentially clipped) blt from the specified buffer to the display. The destination of the presentation blt is the system surface, regardless of which surface is currently being displayed. To blt to the currently displayed member of a flip chain, use PVR2DBlt or PVR2DClippedBlt. The blt uses the cached parameters generated by the PVR2DSetPresentBltProperties call and does no format conversion. The source of the blt is assumed to be in the same format as the system surface.

PVR2DCreateFlipChain

```
PVR2DERROR PVR2DCreateFlipChain (
              PVR2DCONTEXTHANDLE hContext,
              unsigned long ulFlags,
              unsigned long ulNumBuffers,
              unsigned long ulWidth,
              unsigned long ulHeight,
              PVR2DFORMAT eFormat,
              long *plStride,
              unsigned long *pulFlipChainID,
              PVR2DFLIPCHAINHANDLE *phFlipChain);
```

Inputs

`hContext`	The context to operate on
`ulFlags`	If PVR2D_CREATE_FLIPCHAIN_SHARED is set, the flipchain can be shared among multiple user processes. If PVR2D_CREATE_FLIPCHAIN_QUERY is set, the call simply returns a handle to a pre-existing flipchain specified by *pulFlipChainID. If PVR2D_CREATE_FLIPCHAIN_QUERY is not set, a new flipchain is created.
`ulNumBuffers`	The number of buffers in the flip chain.
`ulWidth`	The pixel width of the flip chain to create.
`ulHeight`	The pixel height of the flip chain to create.
`eFormat`	The PVR2DFORMAT pixel format of the screen.
`pulFlipChainID`	If PVR2D_CREATE_FLIPCHAIN_QUERY is set, this contains the unique identifier of the flip chain to query. Must be between 0 and the maximum number of flip chains supported by the device.

Outputs

`plStride`	The stride of the flip chain – will be the same for all buffers.
`pulFlipChainID`	If PVR2D_CREATE_FLIPCHAIN_QUERY is not set, this returns the ID of the newly created flip chain. This is only returned if PVR2D_CREATE_FLIPCHAIN_SHARED is set.
`phFlipChain`	The flip chain that has been created

Returns

`PVR2D_OK`	Success
`PVR2DERROR_GENERIC`	The flip chain cannot be created
`PVR2DERROR_INVALID_PARAMETER`	A parameter is invalid
`PVR2DERROR_INVALID_CONTEXT`	The context is invalid

Description

Creates a flip chain with the given dimensions and pixel format. The number of buffers specified are used to allocate/associate buffers for the flip chain. On any given display device the number of flip chains which can be created may be 0 or more. The memory for the flip buffers may have been previously allocated outside of PVR2D. If the PVR2D_CREATE_FLIPCHAIN_QUERY flag is set, this call simply gets a reference to a pre-existing flip chain specified by *pulFlipChainID. If it is not set, a new flip chain is created and a unique flipchain ID will be returned if PVR2D_CREATE_FLIPCHAIN_SHARED is also set. The stride in bytes of the buffers is returned in plStride. In the case of a display device that only supports one flip chain, the width, height and pixel format should be set to the same as the current display mode, as returned from PVR2DGetDeviceInfo.

PVR2DDestroyFlipChain

```
PVR2DERROR PVR2DDestroyFlipChain (
               PVR2DCONTEXTHANDLE hContext
               PVR2DFLIPCHAINHANDLE hFlipChain);
```

Inputs

`hContext`	The context to operate on.
`hFlipChain`	The flip chain to destroy.

Outputs

Returns

`PVR2D_OK`	Success
`PVR2DERROR_GENERIC`	The flip chain cannot be destroyed
`PVR2DERROR_INVALID_PARAMETER`	The flip chain handle is invalid
`PVR2DERROR_INVALID_CONTEXT`	The context is invalid

Description

Destroys resources associated with a flip chain. The memory for the flip buffers may have been previously allocated outside of PVR2D in which case this call simply breaks the association between the buffers and the flip chain.

PVR2DSetPresentFlipProperties

```
PVR2DERROR PVR2DSetPresentFlipProperties (
               PVR2DCONTEXTHANDLE hContext,
               PVR2DFLIPCHAINHANDLE hFlipChain,
               unsigned long ulPropertyMask,
               long lDstXPos,
               long lDstYPos,
               unsigned long ulNumClipRects,
               PVR2DRECT *pClipRects,
               unsigned long ulSwapInterval);
```

Inputs

`hContext`	The context to operate on
`hFlipChain`	The flip chain on which to set properties. If this is NULL, the properties apply to the present blt.
`ulPropertyMask`	A mask of properties to set. Any of PVR2D_PRESENT_PROPERTY_DSTPOS, PVR2D_PRESENT_PROPERTY_CLIPRECTS PVR2D_PRESENT_PROPERTY_INTERVAL may be Or'd in.
`lDstXPos`	The X position for the flip relative to the system surface origin.
`lDstYPos`	The Y position for the flip relative to the system surface origin.
`ulNumClipRects`	The number of clip rectangles specified in the following parameter
`pClipRects`	A pointer to a list of rectangles describing the un-occluded areas of the surface to be presented.
`ulSwapInterval`	The number of display update periods to wait before a flip takes place.

Outputs

Returns

`PVR2D_OK`	Success
`PVR2DERROR_INVALID_PARAMETER`	A parameter is invalid
`PVR2DERROR_INVALID_CONTEXT`	The context is invalid

Description

Sets the properties for a presentation flip. May be called any time after the creation of a flip chain – will take effect on the next flip call. The ulPropertyMask parameter allows any combination of the present properties to be set at once, by OR'ing the relevant bit into the mask and then supplying the parameter. In the case of a display device that only supports one flip chain, the dest position and cliprects parameters will be ignored.

PVR2DGetFlipChainBuffers

```
PVR2DERROR PVR2DGetFlipChainBuffers (
               PVR2DCONTEXTHANDLE hContext,
               PVR2DFLIPCHAINHANDLE hFlipChain,
               unsigned long *pulNumBuffers,
               PVR2DMEMINFO **ppsMemInfo);
```

Inputs

`hContext`	The context to operate on
`hFlipChain`	The flip chain for which to get buffers

Outputs

`pulNumBuffers`	The number of buffers returned.
`ppsMemInfo`	A list of buffers associated with this flip chain. The number of buffers in this list is the number returned in pulNumBuffers.

Returns

`PVR2D_OK`	Success
`PVR2DERROR_INVALID_PARAMETER`	The flip chain is invalid
`PVR2DERROR_INVALID_CONTEXT`	The context is invalid

Description

Gets the buffers associated with a flip chain.

PVR2DPresentFlip

```
PVR2DERROR PVR2DPresentFlip (
               PVR2DCONTEXTHANDLE hContext,
               PVR2DFLIPCHAINHANDLE hFlipChain,
               PVR2DMEMINFO *psMemInfo,
               long lRenderID);
```

Inputs

`hContext`	The context to operate on
`hFlipChain`	The flip chain to flip from.
`psMemInfo`	The buffer to flip to.
`lRenderID`	Private data associated with this flip

Outputs

Returns

`PVR2D_OK`	Success
`PVR2DERROR_INVALID_PARAMETER`	A parameter is invalid
`PVR2DERROR_DEVICE_UNAVAILABLE`	Device cannot complete request
`PVR2DERROR_INVALID_CONTEXT`	The context is invalid

Description

The present properties are used to perform a (potentially clipped, sized) flip from the specified buffer to the display. The buffer must be one that has been retrieved from PVR2DGetFlipChainBuffers.

PVR2DBlt3D

```
PVR2DERROR PVR2DBlt3D (const PVR2DCONTEXTHANDLE hContext, const
               PPVR2D_3DBLT pBlt3D);
```

Inputs

`hContext`	The context to operate on
`pBlt3D`	PVR2D_3DBLT structure describing the 3D Blt.

Outputs

Returns

`PVR2D_OK`	Success
`PVR2DERROR_INVALID_PARAMETER`	A parameter is invalid
`PVR2DERROR_DEVICE_UNAVAILABLE`	Device cannot complete request
`PVR2DERROR_INVALID_CONTEXT`	The context is invalid

Description

The PVR2DBlt3D function enables PVR2D blts to be routed via the 3D core for the purpose of enhanced functionality, including scaling and custom pixel shader blts.

The PVR2D_3DBLT structure is described below :

```
typedef struct
{
        PVR2D_SURFACE           sDst;
        PVR2D_SURFACE           sSrc;
        PVR2DRECT               rcDest;
        PVR2DRECT               rcSource;
        PVR2D_HANDLE            hUseCode;
        unsigned long           UseParams[2];

} PVR2D_3DBLT, *PPVR2D_3DBLT;
```

`sDst`	Destination surface
`sSrc`	Source surface
`rcDest`	Destination rectangle
`rcSource`	Source rectangle (scaling is enabled if there is a difference in size between the source and dest rectangles)
`hUseCode`	Handle to the custom shader program loaded via the PVR2DLoadUseCode function.
`UseParams[2];`	Optional per-blt constants for the custom shader program.

The register assignments for the USSE pixel shader code are as follows :

`input`	`pa0`	source pixel
`input`	`pa1`	destination pixel as input
`input`	`sa0`	per-blt constant `UseParams[0]`
`input`	`sa1`	per-blt constant `UseParams[1]`
`output`	`o0`	destination pixel output

Power implications of using PVR2DBlt3D

Some PowerVR SGX cores, such as SGX535, have a dedicated low-power 2D core, and in those cases there are power implications when calling the PVR2DBlt3D function. Using the PVR2DBlt3D API will take the 3D core out of low power state and normal 3D core power consumption can be assumed. For cores that do not have a dedicated 2D core then all blts would already be using the 3D core and so the PVR2DBlt3D function would not increase power consumption.

PVR2DBlt3DExt

```
PVR2DERROR PVR2DBlt3DExt (const PVR2DCONTEXTHANDLE hContext, const
               PPVR2D_3DBLT_EXT pBlt3D);
```

Inputs

`hContext`	The context to operate on
`pBlt3D`	PVR2D_3DBLT_EXT structure describing the 3D Blt.

Outputs

Returns

`PVR2D_OK`	Success
`PVR2DERROR_INVALID_PARAMETER`	A parameter is invalid
`PVR2DERROR_DEVICE_UNAVAILABLE`	Device cannot complete request
`PVR2DERROR_INVALID_CONTEXT`	The context is invalid

Description

The PVR2DBlt3DExt function is an extended version of PVR2DBlt3D with enhanced functionality.

The PVR2D_3DBLT_EXT structure is described below :

```
typedef struct
{
        PVR2D_SURFACE           sDst;
        PVR2DRECT               rcDest;
        PVR2D_SURFACE           sSrc;
        PVR2DRECT               rcSource;
        PPVR2D_SURFACE          pSrc2;
        PVR2DRECT*              prcSource2;
        PVR2D_HANDLE            hUseCode;
        unsigned long           UseParams[2];
        unsigned int            uiNumTemporaryRegisters;
        PVR2D_BOOL              bDisableDestInput;

} PVR2D_3DBLT_EXT, *PPVR2D_3DBLT_EXT;
```

`sDst`	Destination surface
`rcDest`	Destination rectangle
`sSrc`	Source surface
`rcSource`	Source rectangle (scaling is enabled if there is a difference in size between the source and dest rectangles)
`pSrc2`	Optional second source surface, NULL if none
`prcSource2`	Second source rectangle, NULL if none

`hUseCode`	Handle to the custom shader program loaded via the PVR2DLoadUseCode function.
`UseParams[2];`	Optional per-blt constants for the custom shader program.
`uiNumTemporaryRegisters`	Number of temporary registers needed by the custom shader program.
`bDisableDestInput`	Set to PVR2D_TRUE if the destination surface is not an input to the blt (this improves the performance of source copy blts).

The register assignments for the USSE pixel shader code are as follows :

`input`	`pa0`	source pixel
`input`	`pa1`	destination pixel as input
`input`	`pa2`	source2 pixel (optional)
`input`	`sa0`	per-blt constant `UseParams[0]`
`input`	`sa1`	per-blt constant `UseParams[1]`
`temporary`	`r0, r1 etc.`	temps as required
`output`	`o0`	destination pixel output

Power implications of using PVR2DBlt3DExt

Some PowerVR SGX cores, such as SGX535, have a dedicated low-power 2D core, and in those cases there are power implications when calling the PVR2DBlt3DExt function. Using the PVR2DBlt3DExt API will take the 3D core out of low power state and normal 3D core power consumption can be assumed. For cores that do not have a dedicated 2D core then all blts would already be using the 3D core and so the PVR2DBlt3DExt function would not increase power consumption.

PVR2DLoadUseCode

```
PVR2DERROR PVR2DLoadUseCode (const PVR2DCONTEXTHANDLE hContext, const
               unsigned char *pUseCode, const unsigned long UseCodeSize,
               PVR2D_HANDLE *pUseCodeHandle);
```

Inputs

`hContext`	The context to operate on.
`pUseCode`	The USSE pixel shader code.
`UseCodeSize`	The number of code bytes to load.

Outputs

`pUseCodeHandle`	Points to a buffer to receive the code handle.

Returns

`PVR2D_OK`	Success
`PVR2DERROR_INVALID_PARAMETER`	A parameter is invalid
`PVR2DERROR_INVALID_CONTEXT`	The context is invalid

Description

The PVR2DLoadUseCode function loads custom pixel shader code into correctly aligned device memory and maps it so that it is accessible to the USSE. The returned handle then can be used as an input to the PVR2DBlt3D and PVR2DBlt3DExt functions.

The custom pixel shader code is compiled from an asm source file using the standard PowerVR useasm.exe and uselink.exe tools, and the resulting code ends up in a C header file as an array of bytes that can be loaded via this function.

PVR2DFreeUseCode

```
PVR2DERROR PVR2DFreeUseCode (const PVR2DCONTEXTHANDLE hContext, const
               PVR2D_HANDLE hUseCodeHandle);
```

Inputs

`hContext`	The context to operate on
`hUseCodeHandle`	Handle to the USSE code bytes to free

Outputs

Returns

`PVR2D_OK`	Success
`PVR2DERROR_INVALID_PARAMETER`	A parameter is invalid
`PVR2DERROR_INVALID_CONTEXT`	The context is invalid

Description

Frees up the memory allocated during a previous call to PVR2DLoadUseCode.

PVR2DWaitForNextHardwareEvent

```
PVR2DERROR PVR2DWaitForNextHardwareEvent(
               const PVR2DCONTEXTHANDLE hContext);
```

Inputs

`hContext`	The context to operate on

Outputs

Returns

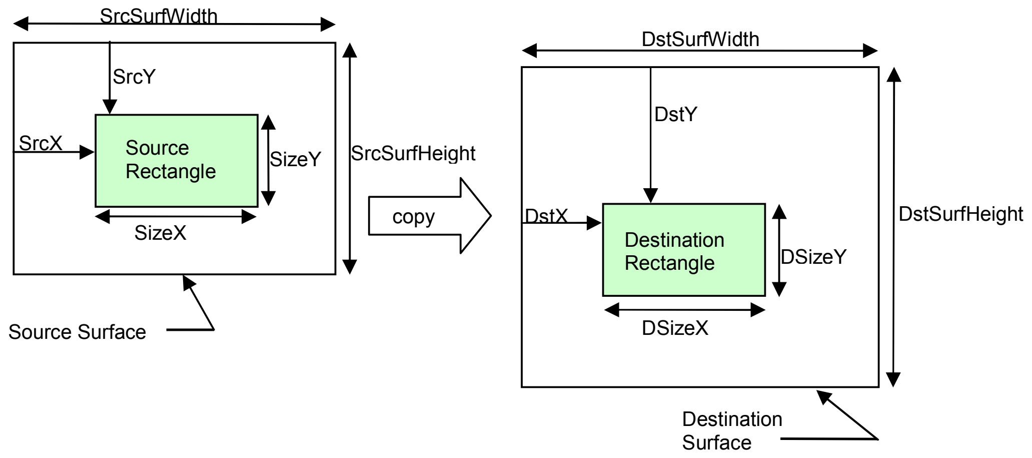
`PVR2D_OK`	Success
`PVR2DERROR_INVALID_CONTEXT`	The context is invalid
`PVR2DERROR_BLT_NOTCOMPLETE`	A platform specific timeout value was reached
`PVR2DERROR_GENERIC_ERROR`	The wait cannot be performed

Description

This API waits for the next significant graphics hardware event to occur. The calling thread will be blocked until the next event occurs or the function times out. Timeout durations if applicable are platform specific.

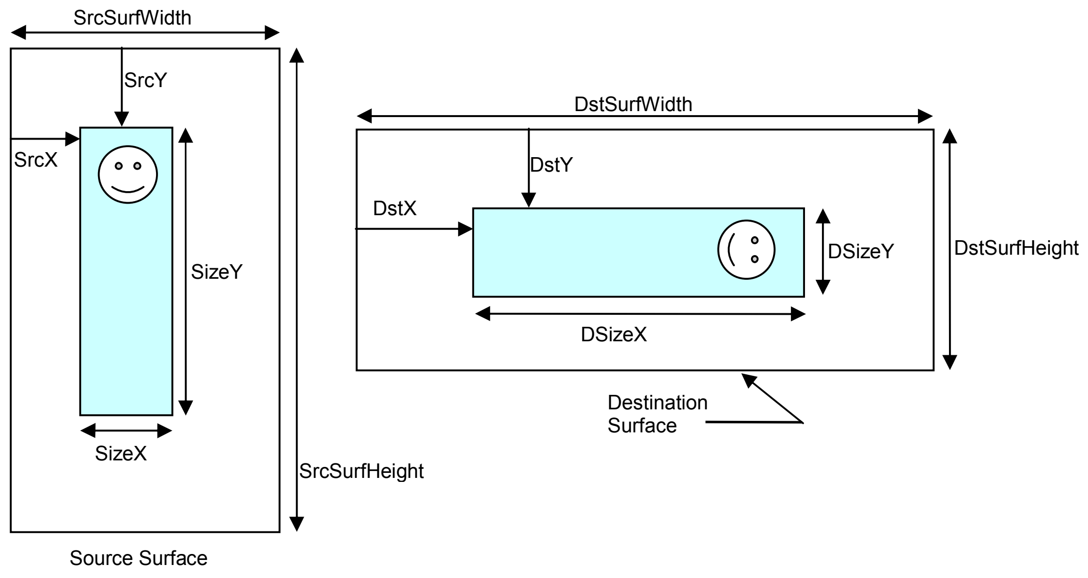
Appendix A. PVR2DBlt rectangles

The following diagram shows how the PVR2DBLTINFO structure fields are used to define the blt rectangles used in the PVR2DBlt function.



Please ensure that the Source Rectangle does not go outside the Source Surface, and the Destination Rectangle does not go outside the Destination Surface. Also note that negative coordinates are not supported. Drawing outside a surface allocation may produce a page fault, in which case the blt will be aborted and the sync object will not complete.

The example below shows a source copy blt with 90 degree rotation :



Note that when rotating by 90 or 270 degrees the caller must ensure that the aspect is also rotated as shown above, so that, for example, the source width SizeX and dest height DSizeY match. The destination surface dimensions must be >= the rotated blt size so that the blt does not draw outside the surface.