SGX Services 4

Software Functional Specification

Filename	:	SGX Services 4.Software Functional Specification.doc
Version	:	4.1.151 External Issue
Issue Date	:	09 Mar 2011
Author	:	POWERVR

Contents

List of Figures

1. Introduction

1.1. Scope

This document describes the high level design of the Services Software components.

1.2. Related Documents

Related Documents
Services 4.0 Software Architectural Specification

2. Functionality

2.1. Summary of Functionality

The following table is a summary of functionality provided by the Services software.

Functionality	Description
Device Initialization	Provides final initialisation of all devices managed by Services
Device Management	Provides initialisation and control of all devices managed by Services as well as Device Class integration and control functions
Memory Management	Provides allocation and mapping for all device addressable memories
Device Functionality	Exposes underlying device functionality
Command Queue Management	Provides mechanisms for asynchronous hardware operation while maintaining synchronisation where necessary
Display Device and Surface Management	Provides integration framework for display device and surface control where the display hardware is managed by 3rd party software
Resource Management	Provides resource management for all consumer services resources
Power Management	Provides power management functionality
Client / Kernel glue	Provides environment agnostic functions to route calls from client to kernel services. Supports Usermode and Kernelmode clients.

3. Software Module Architecture

The relationships and the relative positions of independent services software modules vary depending on the operating system environment. Client drivers of services may be loaded in kernel and/or user space, the figure below illustrating the basic layout:

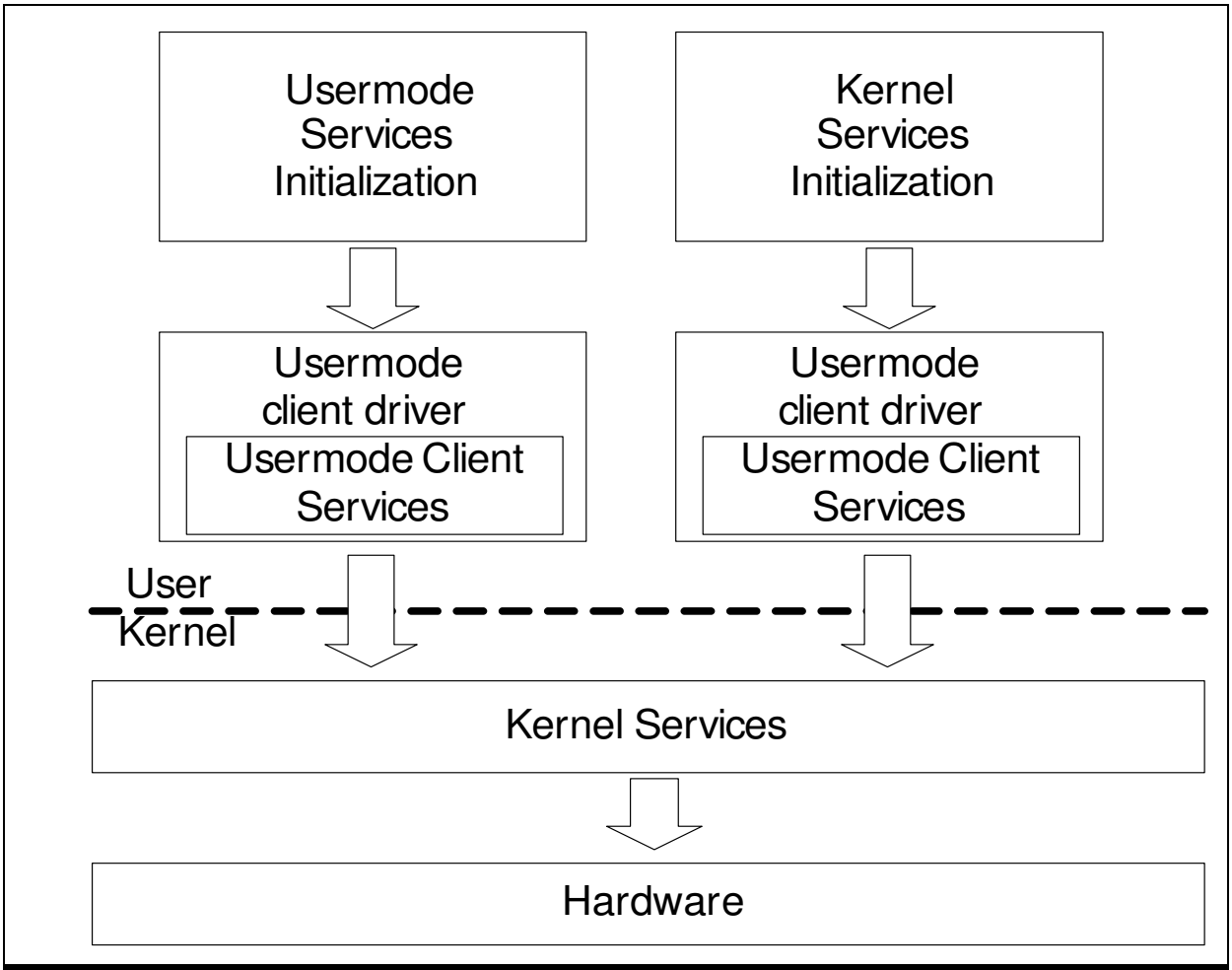

Figure 1 Example of relationship between services software components

4. Services Component Design Overview

As described in previous sections, Services is constructed from independent subcomponents. The following section provides a technical overview of the design of each Services software component.

4.1. System Configuration

Services provides abstraction in the following ways:

- Device – Device control APIs are specified to control a 'Services Managed Device'
- OS – OS abstraction APIs hide Services interactions with the underlying OS
- System – The system layer abstracts the details of a given System (SOC)

A new SOC requires a new system port.

The Services system abstraction is represented by the files in services\system\<socname> directory.

The main function of interest is SysInitialise() in sysconfig.c and responsibilities include:

- Calling OSInitEnvData() – performs (optional) OS specific initialisation
- Setting up psSysData – the central Services control structure
- Calling SysLocateDevices() – locates or specifies each device's ports, registers and memories within the system physical address map. Stores information is static structures for subsequent queries from device specific initialisation code via calls to SysGetDeviceMemoryMap()
- Calling PVRSRVRegisterDevice for each device – registers devices with Services Device Manager
- (optional) overriding device memory context default template (see section 4.4.3)

- (optional) specifying Backing Store Device memory heap associations (see section 4.4.3)
- Calling PVRSRVInitialiseDevice() for each device – requests Services Device Manager to initialise each device.

4.1.1. System dependent address translations

Every system configuration must also provide implementations of pre-defined functions for translating between the following address space types:

- IMG_SYS_PHYADDR – system bus relative physical address
- IMG_CPU_PHYADDR – CPU relative physical address
- IMG_DEV_PHYADDR – Device relative physical address
- IMG_DEV_VIRTADDR – Device Virtual address

Note: Translations to and from IMG_CPU_VIRTADDR are handled by the OS abstraction APIs.

4.2. Device Manager

The Device Manager component has a number of responsibilities which include:

- Device Registration and Initialisation
  The Device Manager provides a set of APIs to be called by the system component to selectively add any device combination to a system configuration. As part of SysInitialise, all devices are first registered with the device manager via a sequence of calls to PVRSRVRegisterDevice(). Next, the system specifies the backing store associations and optionally changes each device's heap attributes. Finally, the devices are initialised by via a sequence of calls to PVRSRVInitialiseDevice().
- Device Abstraction Functions
  Each Services managed device must implement a set of pre-defined APIs, allowing Services to treat all Services devices in a generic fashion. Functions include:
  - Device specific initialisation (and deinitialisation)
  - Device specific LISR call-back
  - Device specific MISR call-back
  - Memory management, abstracting device specific MMU setup and control
- Device Enumeration and Initialisation Client APIs
  The Services API provides a set of APIs that allow client drivers to enumerate devices present in a given Services configuration as well as initialising the device so it can controlled via the device-specific Services APIs.

The Device Manager maintains a linked-list of registered devices and stores each device's information in a 'Device Node' structure, PVRSRV_DEVICE_NODE. The device node hides the device specific details and allows the device manager to treat each device with a common set of handling functions.

with the device manager the Services Initialization perfomes final step to

4.3. Services Initialization

After the Device Manager performs SysInitialise and all devices are registered, the Services Initialization prepares the device configuration and calls the Device Manager to apply it to the device. All resources allocated by the Services Initialization are detached from the calling process and passed to the Kernel Resource Manager.

4.4. Device Memory Management

Device Memory Management is one of the most complex components within Services. Device memory management is concerned with the management of memory addressed by 'devices' which themselves are managed by Services. Device memory does not necessarily mean dedicated local device memory like the DDR RAM on a PCIe graphics cards; it can also include 'system' memory, addressable by a device (an example of a UMA system).

4.4.1. Address Space Types

For device addressable memory Services manages CPU and Device address spaces. For CPU there are the following independent types:

- PVRSRV_HAP_KERNEL_ONLY – only the kernel services driver has CPU access to memory allocated with this attribute
- PVRSRV_HAP_SINGLE_PROCESS – only the process that requested to create the allocation has CPU access to it (generally includes kernel access provided it's in the owning process calling context)
- PVRSRV_HAP_MULTI_PROCESS – kernel services driver has CPU access as well as any client process (mapped on demand unless shared memory architecture OS like WinCE5.0)

Additional CPU related attributes include:

- PVRSRV_HAP_CACHED – request CPU cached memory
- PVRSRV_HAP_UNCACHED – request CPU uncached memory
- PVRSRV_HAP_WRITECOMBINE - request write combined memory (or equivalent write burst technology)

Device addresses are analogous to CPU address types:

- DEVICE_MEMORY_HEAP_PERCONTEXT – only the process that requested to create the allocation has 'Device' access to it
- DEVICE_MEMORY_HEAP_KERNEL – only the Kernel services has 'Device' access to memory of this type
- DEVICE_MEMORY_HEAP_SHARED – an allocation of this type is Device accessible from any process (and kernel) that has a valid device memory address space for the device (see Device Memory Contexts and Heaps)
- DEVICE_MEMORY_HEAP_SHARED_EXPORTED – the same as DEVICE_MEMORY_HEAP_SHARED but the heap information is passed to client drivers so they can make allocations from it. To be deprecated.

The Services framework defines 5 independent address space types to support address space translation and management across varying system architectures:

- IMG_SYS_PHYADDR – system bus relative physical address
- IMG_CPU_PHYADDR – CPU relative physical address. Generally, this is the same as IMG_SYS_PHYADDR but some systems do allow for address offsets from the system bus address per CPU.
- IMG_CPU_VIRTADDR – CPU Virtual address. If the CPU(s) has an MMU and is enabled then IMG_CPU_VIRTADDR represents the addresses that the CPU accesses. If the MMU is not present or disabled then IMG_CPU_VIRTADDR == IMG_CPU_PHYADDR.
- IMG_DEV_PHYADDR – Device relative physical address. Generally, in a UMA system IMG_DEV_PHYADDR == IMG_SYS_PHYADDR. In a non-UMA system there is local device memory and IMG_DEV_PHYADDR != IMG_SYS_PHYADDR unless the local device memory is based in the system address map at 0 (local device memory is generally based at 0 with respect to device memory accesses)
- IMG_DEV_VIRTADDR – Device Virtual Address. If the device has an MMU and is enabled then IMG_DEV_VIRTADDR represents the address that the device actually accesses. If the MMU is not present or disabled then IMG_DEV_VIRTADDR == IMG_DEV_PHYADDR

In addition to the distinction between CPU and Device address spaces and their access attributes there is also the consideration of 'Backing Store'. A 'Backing Store' represents a source of physical memory that is addressable by a device and whose basic unit is a 4 Kbyte physical page. There two basic types of backing store:

- Contiguous Backing Store – usually used to manage the physical memory local to one or more devices (non-UMA) or a contiguous block of system reserved outside the control of the OS (UMA).
- OS Page allocator – OSAllocPages OS abstraction function provides an API to allocate N pages of physical memory from the OS kernel heap. The memory may or may not be physically contiguous.

Note: Services can function with any combination of backing stores in a given configuration assuming the OS and system architecture also support it.

4.4.2. Device Memory Contexts and Heaps

Before a device can access memory it must create a 'device memory context' and then allocate memory from a heap within the device memory context.

A Services managed device may or may not support multiple concurrent device memory contexts in the Services code – to clarify, concurrent meaning device memory contexts existing concurrently rather than being accessed concurrently by the device. If a device only supports one device memory context then the creation of multiple application specific device memory contexts becomes a redundant operation and a handle to the kernel context is always returned. In either case a 'kernel' device memory context is created at boot time or prior to the creation of the first application device memory context (depending on the device and what it might need to do at boot time).

As part of creating an application device memory context Services also creates heaps within the device memory context. These heaps subdivide the address space according to the devices requirements, allowing devices to control where specific allocation types reside and what attributes are applied. Some devices' subcomponents have limited addressing apertures with the device memory context and the existence of heaps ensures that these allocation types are never 'out of range' in terms of device access.

The sequence of calls from a client driver might be:

- PVRSRVCreateDeviceMemContext() - Create a device memory context on a specified device (psDevData). A handle to the device memory context is returned as well as a heap information structure containing a list of heap handles with which allocations can be made
- PVRSRVAllocDeviceMem – Allocate memory from a specified heap by passing the heap handle. Returns a client memory information structure which contains a CPU virtual address (for CPU read/write access) and a device virtual address (for device read/write access)

4.4.3. System Configuration and Customisation

When a device is initialised by Services the device specific initialisation code creates a default template of how to create device memory contexts. This includes information about which heaps exist and what their attributes are. Following device initialisation (in SysInitialise, sysconfig.c) a given system configuration can override the default template, re-specifying the device heap configurations as required.

As well as overriding the default heap template, the system can also create any number of physical Backing Stores and associate them with each device's heaps appropriately (device specific code has no knowledge about the system configuration it may be in).

If no backing stores are configured then Services will assume a UMA system and allocate all device addressable physical memory from an OS heap (OSAllocPages). Alternatively, a system may create N Backing Store heaps and each heap within a device memory context may be associated with any one of the N Backing Stores. There is also a device specific Backing Store to specify where a device's MMU page-tables are sourced.

4.5. Resource Manager

The Resource Manager is used to free-up hardware and software resources created for processes (applications) that have either been killed or shutdown uncleanly.

Whenever a new process (application) connects to Services it is registered with the Resource Manager via a call to `ResManProcessConnect`(). This function creates a process specific resource item list to which resources can be attached as they are created.

As resources are created for a process, Services registers the resource items which are attached to the resource manager's process specific resource item list. As part of registering a resource item the following information is passed to the function `ResManRegisterRes`():

- Resource Type – used to differentiate between the different resources available in common Services and device specific Services code.
- Resource Type specific clean-up call-back function
- Up to two arguments to be passed to the clean-up call-back function

- The process ID

Whenever an application shuts downs (cleanly or not) the process resource item list is cleaned-up via a call to `ResManProcessDisconnect`(). The resources are clean-up sequentially, the exact order specified by their Resource Type. Clean-up is performed by calling the resource item call-back stored in each Resource item information structure.

The implementation of the clean-up call-back function varies depending on the resource type. The responsibilities of each call-back implementation are twofold:

- The function must ensure that all hardware and/or software operations on the resource to be cleaned-up have been flushed and, if not, take appropriate action to forcefully flush operations or abort all pending operations. In either case, the synchronisation state of the resource must be brought to the 'idle' state (no operations pending)
- Once the resource has been brought to the idle state the function frees the software and/or hardware resource(s).

When all the process resources items have been freed the Resource Manager exits.

4.6. Kernel Resource Manager

The Kernel Resource Manager is used to free-up hardware and software resources created by the Services Initialization. Those resources are detached from the initialization process and should remain available until the driver is unloaded.

The `KernelResManProcessConnect`() function creates a specific resource item list to which kernel resources can be attached as they are detached from the initialization context.

As resources are created by the Services Initialization stage they are firstly detached from the per process Resource Manager to prevent them from being destroyed when the process disconnects. Then they are registered with the resource manager's kernel specific resource item list. As part of registering a resource item the following information is passed to the function `KernelResManRegisterRes`():

- Resource Type – used to differentiate between the different resources available in common Services and device specific Services code.
- Resource Type specific clean-up call-back function
- Up to two arguments to be passed to the clean-up call-back function

Whenever the driver is being unloaded, kernel resource item list is cleaned-up via a call to `KernelResManProcessDisconnect`(). The resources are clean-up sequentially, the exact order specified by their Resource Type. Clean-up is performed by calling the resource item call-back stored in each Resource item information structure.

The implementation of the clean-up call-back function varies depending on the resource type. The responsibilities of each call-back implementation are:

- The function must ensure that all hardware and/or software operations on the resource to be cleaned-up have been flushed and, if not, take appropriate action to forcefully flush operations or abort all pending operations. In either case, the synchronisation state of the resource must be brought to the 'idle' state (no operations pending)
- Once the resource has been brought to the idle state the function frees the software and/or hardware resource(s).

When all the kernel resources items have been freed the Resource Manager exits

4.7. Low and Medium ISRs (L/MISR)

Services provides an interrupt abstraction framework to reduce effort in porting to varying OS, system(SOC) and device combinations, avoiding custom code and associated maintenance costs.

The LISR is analogous to standard OS interrupt Services Routines (ISR) running with high priority, low latency and small workload. The Services code base supports up to one 'Device LISR' per device in a given system configuration. Alternatively, there is the 'System LISR' to be used in systems where devices share the same hardware interrupt and in this case all devices effectively use the same LISR.

The MISR runs with lower priority, higher latency and larger workload than an LISR. The underlying MISR implementation can vary depending on the OS (e.g. IST, DPC, tasklet etc.) and it is implemented within the Services OS abstraction code.

Whether a given system uses N x 'Device LISRs' or one 'System LISR' there is only one MISR in any system configuration and it is scheduled by all LISRs when they have finished processing. The MISR is primarily used to perform Software Command Queue Processing (see Software Command Queues).

Each Services managed device registers LISR and MISR call-back functions. The LISR device call-backs are generally concerned with hardware interrupt handling (identification and clearing) as well as synchronisation object updates. The MISR device call-backs are reserved for more specialised use cases such as Hardware Recovery. Note: MISR per-device call-backs are always called from a single MISR.

When porting Services to a new OS then the OS must provide functionality equivalent to one or more LISRs and a single MISR in order to successfully implement the 'OS port'.

4.8. Hardware Recovery

Hardware Recovery is a fallback mechanism to recover from device errors which would otherwise cause the system to lock up. The implementation details are device-specific, but a description of the general process is given here.

There are two code paths in which device errors can be detected, although in practice the two paths should share much of their code.

The fast detection code path is implemented in a timer task running on the device's microkernel, at a relatively high frequency. This task is responsible for querying the values of signature registers for all device hardware modules which are known to be active at the time. By storing the values of these signatures and checking whether any have changed over a period of time, hardware lockups can be detected. In this case an interrupt is generated back to the host to specify that hardware recovery processing should be executed, which is done in the device's MISR call-back routine.

The slow path is implemented in a timer task running on the host processor, at a relatively low frequency. This is required in order to recover from serious device errors which prevent the microkernel timer task itself from executing, such as page faults. The host timer's algorithm is similar to the microkernel timer, except that it is checking whether the number of timer tasks being scheduled on the microkernel is increasing, effectively treating the cumulative number of microkernel tasks scheduled as a signature register. Again, if a lockup is detected, the hardware recovery code is executed.

When hardware recovery processing is invoked, the following needs to be done from the host:

If required, any diagnostic hardware registers are read and dumped out for debug purposes

The device is reset to clear the error condition

A signal is sent to the microkernel to indicate that it should perform cleanup of its own data structures after the reset

After the microkernel has completed its dummy processing, re-process the software queues to flush any outstanding commands

The following cleanup is required on the device's microkernel:

Detect any host kick events which may have been lost while the device was locked up, and process them by reading the kernel CCB

Dummy-process any tasks which were being processed or were outstanding at the time of the lockup, including updating synchronisation objects and status values

Re-initialise any data structures as required

Re-initialise any hardware state as required

Signal back to the host that recovery is complete and new tasks may be submitted to the device

4.9. Software Command Queues

Software Command Queues are general purpose circular buffers into which private commands may be inserted and subsequently processed in an asynchronous fashion. Command queue creation, destruction and command insertion are always done in the kernel driver in response to synchronous

requests from user mode processes. Command processing may occur in one of two places: immediately after command insertion (synchronous) or executed within the MISR (asynchronous). In either case, the order processing of commands is governed by the following rules:

- Within a given queue commands are processed in the original order of submission
- All queues are stored in a singly linked list and processed sequentially in the order in which the list is constructed
- The processing of a given command depends on the state of its associated synchronisation objects (see Figure 2) where each object is used to manage the order of read/write operations on discrete buffers. A command cannot be processed until the synchronisation objects have reached a specified state (stored in the command itself). As well as enforcing the order of operations on discrete buffers, synchronisation objects indirectly control the order of execution of commands from different queues, providing an effective cross-queue synchronisation mechanism.

Figure 2 shows a list of two command queues: Q1 and Q2. The queues are implemented as circular buffers whose sizes are specified when they're created. A Read and Write Offset are used to manage space in the queue as well as specifying the command insertion and removal positions.

The example presented in the figure uses one queue for '2D Blits' and one queue for 'Display Flips'. The blit and flip commands refer to Surfaces 1 through 4, each of which has an associate synchronisation object for sequencing read/write accesses.

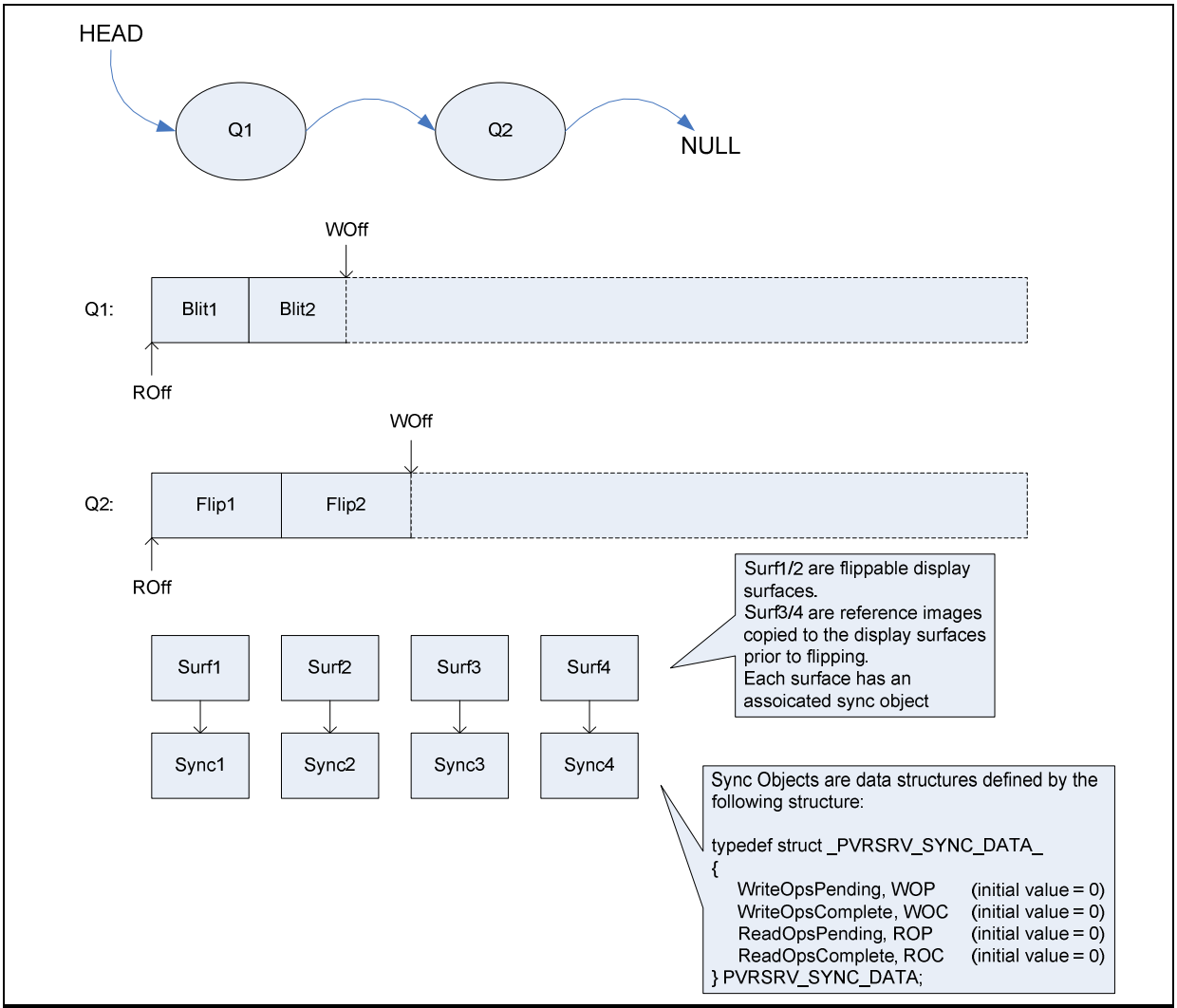

Figure 2 Command Queues

Figure 3 describes the steps involved in inserting commands into queues. The first command to be queued is Blit1 which copies pixels from image surface 3 to display surface 1. A snapshot of the

synchronisation state of both surfaces is copied into the command and made a dependency of the command's execution, i.e. command processing will be blocked until the WOC and ROC have reach the snapshot values stored in the command (WOPd and ROPd).

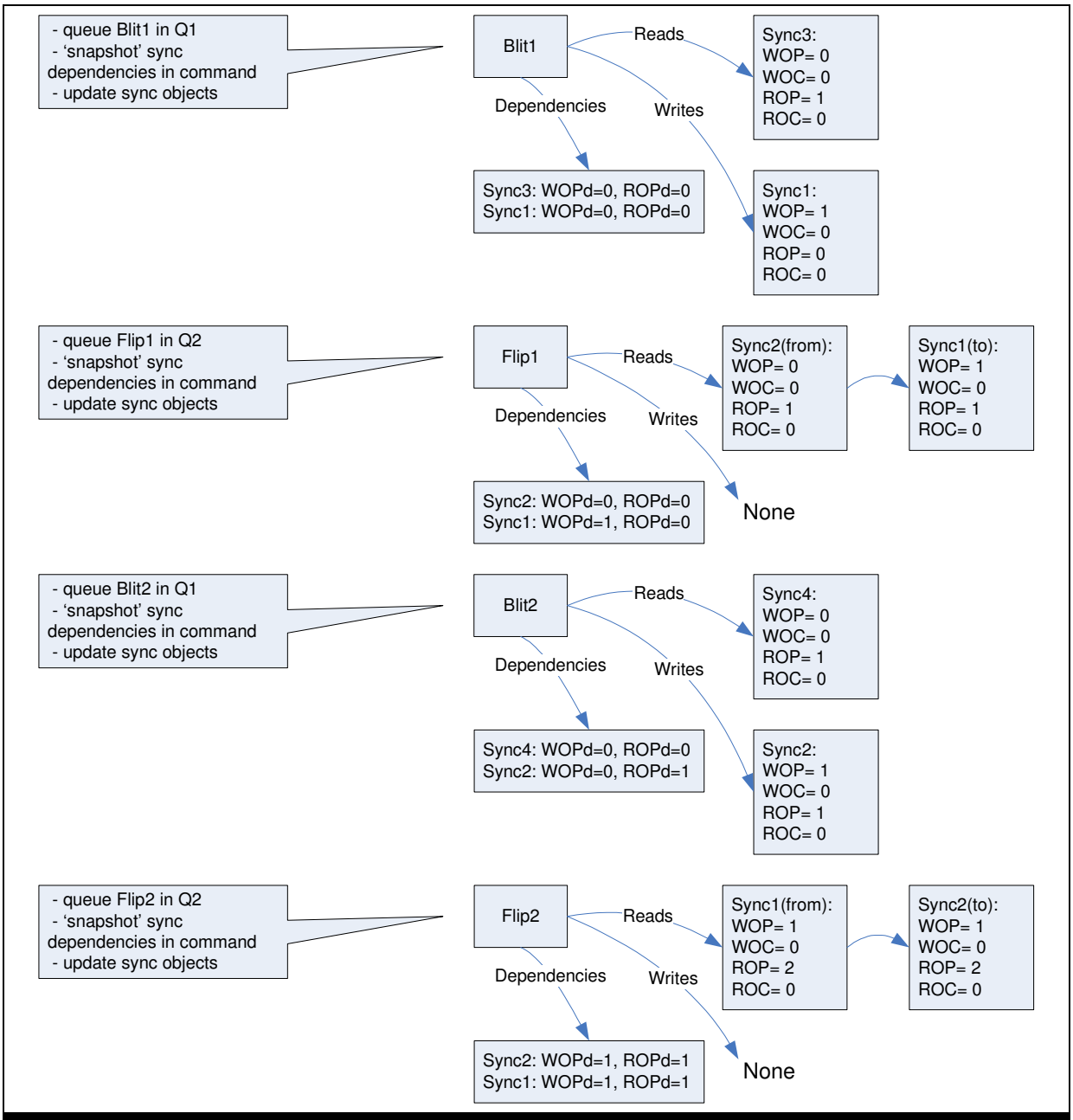

Figure 3 Command Insertion

The commands in the same queue will always be processed in the order in which they were submitted. Commands spread across more than one queue do not necessarily need to be processed in the order of submission rather operations on surfaces need to be processed in the order of submission.

Figure 4 and Figure 5 describe command processing operations. Command processing occurs on the MISR or immediately following command insertion. The MISR is scheduled by an LISR which itself is scheduled following a hardware event signalling the potential completion of a command, e.g. Vsync Interrupt, 3D render complete, 2D blit complete etc. There is only one instance of the MISR irrespective of the number of LISRs.

Assume there has been a hardware event resulting in the MISR performing command processing. The command processor will try to process all the commands from a single queue before processing the next queue. If a command fails its dependency check then the command processor moves to the next queue. Once the command processor has tried to process all queues once it will exit.

In Figure 4 the command process starts by trying to process Q1 and finds Blit1 command. The Blit1 command has stored dependency values (sync3: WOPd, ROPd, sync1: WOPd, ROPd) against which it must match the synchronisation object WOC and ROC values. Blit1 passes the dependency check and blit operation is started on the hardware. The command processor fails to process Blit2 because there is a read outstanding on Sync2. Similarly, Q2, Flip1 also fails but due to a write dependency – Blit1 is still running.

At some point Blit1 complete event occurs and schedules an LISR where the synchronisation object(s) is updated to reflect the completed operation. The LISR then schedules the MISR, where the command processor is attempts to process Flip1 which also fails the dependency check.

When the next vsync interrupt occurs it unblocks the next blit command. The next blit complete unblocks the next flip and so on.

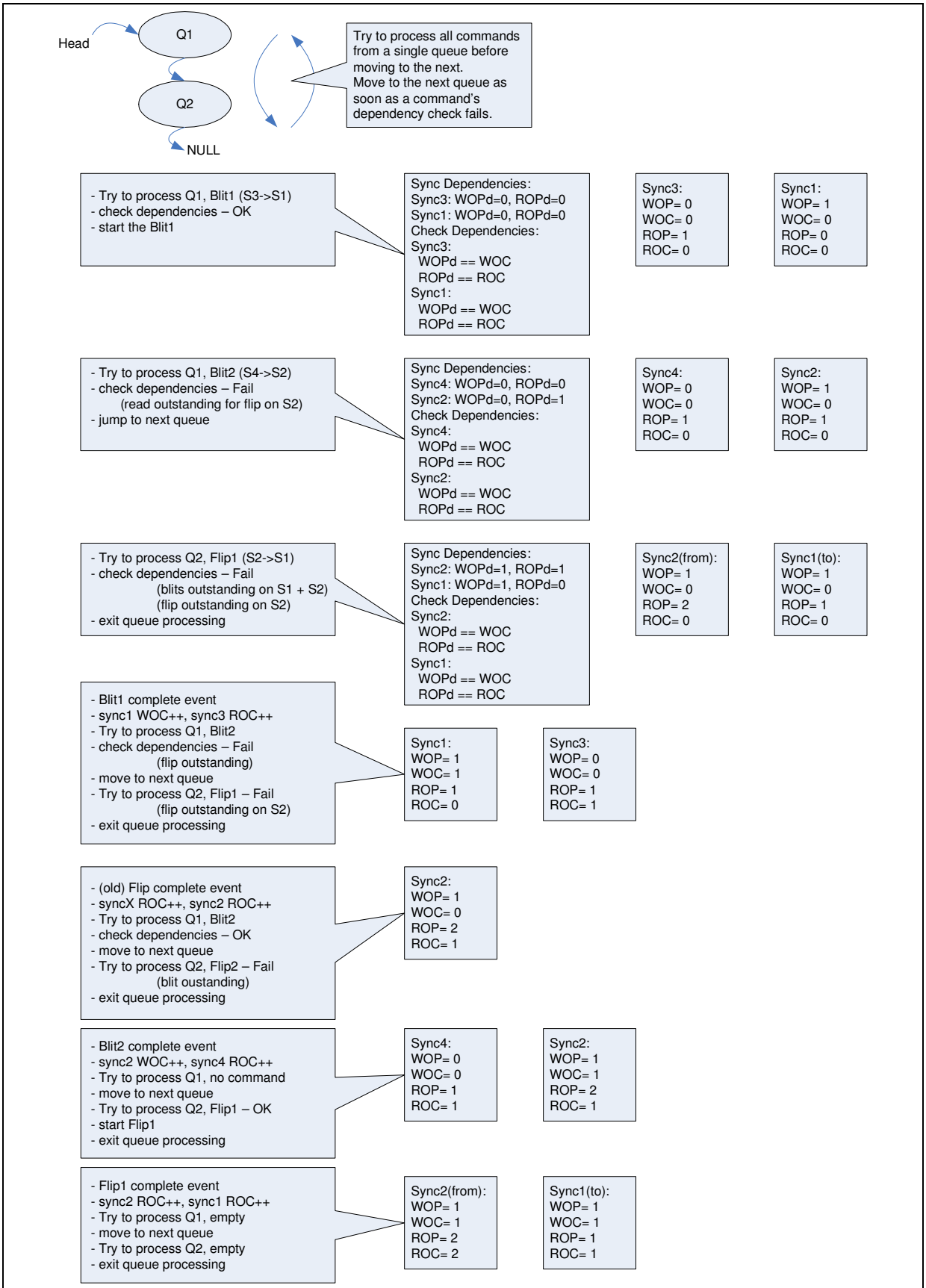

Figure 4 Command Processing

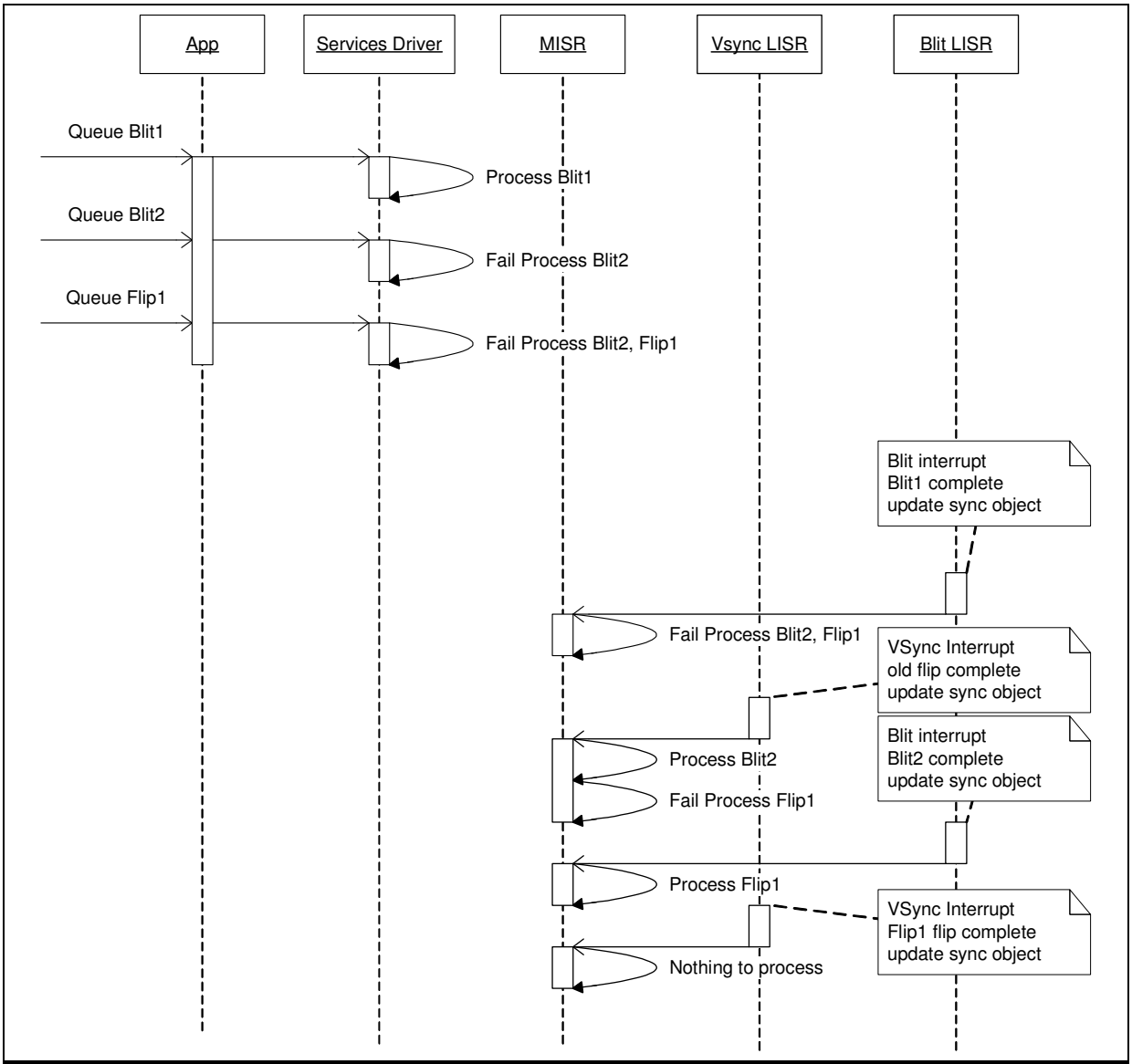

Figure 5 Command Processing Sequence Diagram

4.10. Device Class API

The Device Class API abstracts the control of Display and Buffer Devices. Display devices cover 'display controllers' responsible for reading pixel data memory for one or more display panels. Buffer devices cover a broader range of devices, including cameras and video decodes. The common feature of Buffer devices is that they are 'sources' (or 'producers') of pixel data.

See 3rd Party Display/Buffer API documents for more details.

5. Kernel Services Functionality

The Kernel Services is made up of distinct components, each of which is responsible for different areas of functionality. It should be noted that there are numerous internal interfaces that are not documented here since they do not require any modification during customer platform porting.

5.1. Operating System Interface

Note: most OS abstraction APIs are prefixed 'OS' (formerly 'Host') but some are prefixed 'PVRSRV'. The distinction is that 'OS' prefix represents internal OS abstraction APIs (only called by services and

client services code) whereas 'PVRSRV' prefix represents OS abstraction APIs that also are available to client drivers, e.g. OGLES. Providing OS abstraction to client drivers results in nearly all porting tasks being confined to Services.

See the Services Porting Guide for API details

5.2. System (SOC) Interface

5.2.1. Sysconfig.c

This file contains a group of functions prefixed with 'sys'. The sys functions are called by kernel services and are common to all SOC implementations. Function prototypes can be found in `system/include/syscommon.h`.

SysInitialise

```
PVRSRV_ERROR SysInitialise(IMG_VOID)
```

This function initialises the whole of kernel services at 'driver load' time. Responsibilities include:

- Allocation and/or initialisation of gsSysData, the top-level data structure in kernel services
- Allocation and initialisation of OS/environment specific data, if required (see OSInitEnvData)
- Specification of services managed devices present in the System (SOC)
- Specify each device's physical address map (ports, registers, etc) (see SysLocateDevices)
- Carries out device registration
- (Optionally) modifies each device's default memory configuration
- (Optionally) creates heaps to manage local memory backing stores (non-UMA architecture only)
- (Optionally) Specifies physical memory backing store types for all devices' memory heaps
- Carries out device initialisation

  RETURN
  - success - PVRSRV_OK
  - failure – PVRSRV_ error code

SysDeinitialise

```
PVRSRV_ERROR SysDeinitialise (SYS_DATA *psSysData)
```

This function de-initialises the whole of kernel services at 'driver unload' time

IN – psSysData – system data, setup by SysInitialise

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

SysLocateDevices

```
PVRSRV_ERROR SysLocateDevices(SYS_DATA *psSysData)
```

This function specifies each device's physical address map. Each physical address map is stored as a static structure and retrieved at device initialisation time via a call to SysGetDeviceMemoryMap

Note: this function may include dynamic device enumeration, e.g. PCI enumeration.

IN – psSysData – system data, setup by SysInitialise

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

SysGetDeviceMemoryMap

```
PVRSRV_ERROR SysGetDeviceMemoryMap(
       PVRSRV_DEVICE_TYPE eDeviceType,
       IMG_VOID **ppvDeviceMap)
```

This function returns a device address map for the specified device. The device specific initialisation routines will call this function to retrieve the physical addresses of the device's components (registers, ports, etc)

IN – eDeviceType – the device type to return memory map for, e.g. PVRSRV_DEVICE_TYPE_SGX

OUT – ppvDeviceMap – a device specific structure describing the device map

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

SysCpuPAddrToDevPAddr

```
IMG_DEV_PHYADDR SysCpuPAddrToDevPAddr (
       PVRSRV_DEVICE_TYPE eDeviceType, IMG_CPU_PHYADDR CpuPAddr)
```

This function translates a CPU physical address to a device physical address

IN – eDeviceType – device type to distinction between differing device physical address spaces

IN – CpuPAddr – CPU physical address

RETURN – device physical address

SysSysPAddrToDevPAddr

```
IMG_DEV_PHYADDR SysSysPAddrToDevPAddr (
       PVRSRV_DEVICE_TYPE eDeviceType, IMG_SYS_PHYADDR SysPAddr)
```

This function translates a system (Bus) physical address to a device physical address

IN – eDeviceType – device type to distinction between differing device physical address spaces

IN – SysPAddr – CPU physical address

RETURN – device physical address

SysDevPAddrToSysPAddr

```
IMG_SYS_PHYADDR SysDevPAddrToSysPAddr (
       PVRSRV_DEVICE_TYPE eDeviceType, IMG_DEV_PHYADDR DevPAddr)
```

This function translates a device physical address to a system (bus) physical address

IN – eDeviceType – device type to distinction between differing device physical address spaces

IN – DevPAddr – Device physical address

RETURN – device physical address

SysSysPAddrToCpuPAddr

```
IMG_CPU_PHYADDR SysSysPAddrToCpuPAddr (IMG_SYS_PHYADDR sys_paddr)
```

This function translates a System (Bus) physical address to a CPU physical address

IN – sys_paddr – System physical address

RETURN – CPU physical address

SysCpuPAddrToSysPAddr

```
IMG_SYS_PHYADDR SysCpuPAddrToSysPAddr (IMG_CPU_PHYADDR cpu_paddr)
```

This function translates a CPU physical address to a System (Bus) physical address

IN – cpu_paddr – CPU physical address

RETURN – system physical address

SysOEMFunction

```
PVRSRV_ERROR SysOEMFunction ( IMG_UINT32     ui32ID,
                              IMG_VOID       *pvIn,
                              IMG_UINT32     ulInSize,
                              IMG_VOID       *pvOut,
                              IMG_UINT32     ulOutSize)
```

SysOEMFunction is a marshalling function for custom OEM routines. The caller passes an ID (ui32ID) to invoke OEM defined functionality, along with input and output pointers. Allows OEMS to implement arbitrary customisations but without modifying common services code.

SysOEMFunction is accessed by 3rdparty Display Class devices via a kernel services jumptable.

SysSystemPrePowerState

```
PVRSRV_ERROR SysSystemPrePowerState(PVR_POWER_STATE eNewPowerState)
```

System pre-power state transition function

IN – eNewPowerState – new power state

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

SysSystemPostPowerState

```
PVRSRV_ERROR SysSystemPostPowerState(PVR_POWER_STATE eNewPowerState)
```

System post-power state transition function

IN – eNewPowerState – new power state

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

SysDevicePrePowerState

```
PVRSRV_ERROR SysDevicePrePowerState(IMG_UINT32 ui32DeviceIndex,
       PVR_POWER_STATE eNewPowerState, PVR_POWER_STATE eCurrentPowerState)
```

System pre-power state transition function

IN – eNewPowerState – new power state

IN – eCurrentPowerState – current power state

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

SysDevicePostPowerState

```
PVRSRV_ERROR SysDevicePostPowerState(IMG_UINT32 ui32DeviceIndex,
       PVR_POWER_STATE eNewPowerState, PVR_POWER_STATE eCurrentPowerState)
```

System post-power state transition function

IN – eNewPowerState – new power state

IN – eCurrentPowerState – current power state

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

SysGetInterruptSource

```
IMG_UINT32 SysGetInterruptSource(SYS_DATA* psSysData)
```

In some systems, the same interrupt source may correspond to different events from different devices. In such systems, it is necessary to determine if the incoming interrupt should be handled by a given device's interrupt services routine (This requires hardware support such that the software can determine whether a device is the interrupt source, e.g. interrupt status/enable registers).

During system initialisation, the system-specific code in sysconfig.c calls the Services function PVRSRVRegisterDevice, passing the value of the interrupt bit which corresponds to the device being registered. When an interrupt occurs, Services checks whether this interrupt bit is present in the bitmask returned from SysGetInterruptSource. If so, the interrupt is processed. Otherwise, it is ignored.

In a system where the Device has a dedicated interrupt line (i.e. there are no other possible interrupt sources), a safe return value from SysGetInterruptSource would be 0xFFFFFFFF.

Some operating systems provide interrupt filter and dispatch functionality, calling specific driver ISR only when needed. In this case, when code path reaches the SGX driver there is already no doubt that it is to be handled. If an operating system does this then simply implement SysGetInterruptSource to return 0xFFFFFFFF.

SysClearInterrupts

```
IMG_VOID SysClearInterrupts(SYS_DATA* psSysData, IMG_UINT32 ui32ClearBits)
```

SysClearInterrupts clears system(SOC) specific interrupts specified by ui32ClearBits

6. Client Services – 'The Services API'

6.1. Connecting to Services

```
PVRSRV_ERROR IMG_CALLCONV PVRSRVConnect(PVRSRV_CONNECTION **ppsConnection)
```

This function makes the initial connection to kernel services

IN/OUT – psConnection – connection information

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

```
PVRSRV_ERROR IMG_CALLCONV PVRSRVDisconnect(PVRSRV_CONNECTION *psConnection)
```

This function closes a connection to kernel services

IN – psConnection – connection information

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

6.2. Device Enumeration and Initialisation

```
PVRSRV_ERROR IMG_CALLCONV PVRSRVEnumerateDevices( PVRSRV_CONNECTION *psConnection,
        IMG_UINT32 *puiNumDevices,
        PVRSRV_DEVICE_IDENTIFIER *puiDevIDs)
```

This function enumerates all services managed devices

IN – psConnection – connection information

OUT – puiNumDevices – number of devices

OUT – puiDevIDs – device information

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

```
PVRSRV_ERROR IMG_CALLCONV PVRSRVAcquireDeviceData(PVRSRV_CONNECTION *psConnection,
        IMG_UINT32 uiDevIndex,
        PVRSRV_DEV_DATA *psDevData,
        PVRSRV_DEVICE_TYPE eDeviceType)
```

This function retrieves the 'device data' on a specific device, effectively opening a device specific API

IN – psConnection – connection information

IN – uiDevIndex – device identifier

OUT – psDevData – device data

OUT – eDeviceType – device type

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

6.3. Device Specific APIs

6.3.1. SGX

```
PVRSRV_ERROR IMG_CALLCONV SGXGetClientInfo (PVRSRV_DEV_DATA *psDevData,
       PVRSRV_SGX_CLIENT_INFO *psSGXInfo)
```

This function retrieves client information on the SGX device

IN – psDevData – device data

OUT – psSGXInfo – SGX client information

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

```
PVRSRV_ERROR IMG_CALLCONV SGXReleaseClientInfo (PVRSRV_DEV_DATA *psDevData,
       PVRSRV_SGX_CLIENT_INFO *psSGXInfo)
```

This function 'releases' client information on the SGX device

IN – psDevData – device data

OUT – psSGXInfo – SGX client information

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

```
PVRSRV_ERROR IMG_CALLCONV SGXKickTA (PVRSRV_DEV_DATA *psDevData,
       PSGX_KICKTA psKickTA,
       SGX_KICKTA_OUTPUT *psKickTAOutput,
       IMG_PVOID pvKickTAPDUMP,
       IMG_PVOID pvKickSubmit)
```

This function schedules a 'kick' of the TA on the SGX device

IN - psDevData – device data

IN - psKick – kick information

IN – psKickOutput – kick output information

IN – pvKickPDUMP – pdump specific information

IN – psKickSubmit – OS specific argument. Pass IMG_NULL unless OS implements GPU scheduling style graphics sub-system.

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

```
PVRSRV_ERROR IMG_CALLCONV SGXCreateRenderContext(PVRSRV_DEV_DATA *psDevData,
       PSGX_CREATERENDERCONTEXT psCreateRenderContext,
       IMG_HANDLE *phRenderContext,
       PPVRSRV_CLIENT_MEM_INFO *ppsVisTestResultMemInfo)
```

This function creates a render context on the SGX device

IN – psDevData – device data

IN - psCreateRenderContext – Information describing render context to create

OUT – phRenderContext – returned render context handle

OUT – ppsVisTestResultMemInfo – visibility results buffer (used for occlusion queries)

RETURN

- success - PVRSRV_OK

- failure – PVRSRV_ error code

A pointer to a SGX_CREATERENDERCONTEXT structure is passed to the create function and this contains various information, including the size of the 'parameter buffer' to create. The minimum parameter buffer request size for SGX core variants is 40960bytes (10 x 4kbyte pages).

```
PVRSRV_ERROR IMG_CALLCONV SGXDestroyRenderContext(PVRSRV_DEV_DATA *psDevData,
       IMG_HANDLE hRenderContext,
       PVRSRV_CLIENT_MEM_INFO  *psVisTestResultMemInfo)
```

This function destroys a render context on the SGX device

IN – psDevData – device data

IN – hRenderContext –render context handle

IN - ppsVisTestResultMemInfo – visibility results buffer (used for occlusion queries)

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

```
PVRSRV_ERROR IMG_CALLCONV SGXAddRenderTarget (PVRSRV_DEV_DATA *psDevData,
       SGX_ADDRENDTARG *psAddRTInfo,
       IMG_HANDLE *phRTDataSet)
```

This function adds a render target on the SGX device

IN – psDevData – device data

IN - psAddRTInfo – Information describing render target to create

OUT - phRTDataSet – returned render target handle

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

```
PVRSRV_ERROR IMG_CALLCONV SGXRemoveRenderTarget (PVRSRV_DEV_DATA *psDevData,
       IMG_HANDLE hRenderContext,
       IMG_HANDLE hRTDataSet)
```

This function removes a render target on the SGX device

IN – psDevData – device data

IN - hRenderContext – render context handle

IN - hRTDataSet – Render target information handle

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

```
PVRSRV_ERROR IMG_CALLCONV SGXSetContextPriority(PVRSRV_DEV_DATA *psDevData,
       SGX_CONTEXT_PRIORITY *pePriority,
       IMG_HANDLE hRenderContext,
       IMG_HANDLE hTransferContext)
```

Sets the priority of the render and transfer contexts

IN – psDevData – device data

IN – pePriority - priority

IN - hRenderContext – render context handle

IN - hTransferContext – transfer context handle

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

```
PVRSRV_ERROR IMG_CALLCONV SGXGetMiscInfo(PVRSRV_DEV_DATA *psDevData,
       SGX_MISC_INFO *psData)
```

Sets/gets miscellaneous SGX information

IN – psDevData – device data

IN – psData – misc info request structure

Requests are listed in the enumeration SGX_MISC_INFO_REQUEST:

```
        SGX_MISC_INFO_REQUEST_CLOCKSPEED
                Request current SGX core clock speed
        SGX_MISC_INFO_REQUEST_SGXREV
                Request SGX core revision
        SGX_MISC_INFO_REQUEST_DRIVER_SGXREV
                Request SGX driver revision
        SGX_MISC_INFO_REQUEST_MEMREAD
                Request USSE memory read
        SGX_MISC_INFO_REQUEST_SET_HWPERF_STATUS,
                Set HW performance status
        SGX_MISC_INFO_REQUEST_SET_BREAKPOINT
                Set Data Breakpoint
        SGX_MISC_INFO_REQUEST_WAIT_FOR_BREAKPOINT
                Wait for breakpoint
        SGX_MISC_INFO_REQUEST_POLL_BREAKPOINT
                Poll breakpoint
        SGX_MISC_INFO_REQUEST_RESUME_BREAKPOINT
                Resume breakpoint
        SGX_MISC_INFO_DUMP_DEBUG_INFO
                Dump debug Info
        SGX_MISC_INFO_PANIC
                Cause OS 'Panic'
        SGX_MISC_INFO_REQUEST_SPM
                Request SPM stats
        SGX_MISC_INFO_REQUEST_ACTIVEPOWER
                Request active power stats
        SGX_MISC_INFO_REQUEST_LOCKUPS
                Request lockup stats
```

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

6.4. Memory Management

```
PVRSRV_ERROR IMG_CALLCONV PVRSRVCreateDeviceMemContext(PVRSRV_DEV_DATA *psDevData,
        IMG_HANDLE *phDevMemContext,
        IMG_UINT32 *pui32ClientHeapCount,
        PVRSRV_HEAP_INFO *psHeapInfo)
```

This function creates a 'memory context'

IN – psDevData – device data

OUT - phDevMemContext – pointer to memory context handle

OUT - pui32ClientHeapCount – number of heaps returned to the client

OUT – psHeapInfo – pointer to an array of heap information structures

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

```
PVRSRV_ERROR IMG_CALLCONV PVRSRVDestroyDeviceMemContext(PVRSRV_DEV_DATA *psDevData,
        IMG_HANDLE hDevMemContext)
```

This function destroys a 'memory context'

IN – psDevData – device data

IN – hDevMemContext – memory context handle

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

```
PVRSRV_ERROR IMG_CALLCONV PVRSRVAllocDeviceMem(PVRSRV_DEV_DATA *psDevData,
        IMG_HANDLE hDevMemHeap,
        IMG_UINT32 ui32Attribs,
        IMG_UINT32 ui32Size,
        IMG_UINT32 ui32Alignment,
        PVRSRV_CLIENT_MEM_INFO **ppsMemInfo)
```

This function allocates device addressable memory

IN – psDevData – device data

IN – hDevMemHeap – memory heap handle

IN – ui32Attribs - attributes

IN - ui32Size – size in bytes

IN - ui32Alignment – alignment in bytes

OUT – ppsMemInfo – returned memory information structure

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

```
PVRSRV_ERROR IMG_CALLCONV PVRSRVFreeDeviceMem(PVRSRV_DEV_DATA *psDevData,
PVRSRV_CLIENT_MEM_INFO *psMemInfo)
```

This function frees device addressable memory

IN – psDevData – device data

IN – psMemInfo –memory information structure

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

```
PVRSRV_ERROR PVRSRVReserveDeviceVirtualMem(PVRSRV_DEV_DATA *psDevData,
        IMG_HANDLE hDevMemHeap,
        IMG_DEV_VIRTADDR *psDevVAddr,
        IMG_UINT32 ui32Size,
        IMG_UINT32 ui32Alignment,
        PVRSRV_CLIENT_MEM_INFO **ppsMemInfo)
```

This function reserves device virtual memory

IN – psDevData – device data

IN – hDevMemHeap – device memory heap on which to reserve memory

IN – psDevVAddr – user specified device virtual address for the allocation. NULL if specified by services

IN – ui32Size – size of allocation in bytes

IN – ui32Alignment – alignment in bytes

OUT – ppsMemInfo – memory information structure

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

```
PVRSRV_ERROR IMG_CALLCONV PVRSRVFreeDeviceVirtualMem(PVRSRV_DEV_DATA *psDevData,
        PVRSRV_CLIENT_MEM_INFO *psMemInfo)
```

This function frees (previously reserved) device virtual memory

IN – psDevData – device data

IN – psMemInfo – memory information structure

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

```
PVRSRV_ERROR PVRSRVMapDeviceMemory (PVRSRV_DEV_DATA *psDevData,
        PVRSRV_CLIENT_MEM_INFO *psSrcMemInfo,
        IMG_HANDLE hDstDevMemHeap,
        PVRSRV_CLIENT_MEM_INFO **ppsDstMemInfo)
```

This function maps an existing allocation to the specified device virtual heap

IN – psDevData – device data

IN – psSrcMemInfo – memory information structure

IN – hDstDevMemHeap – handle to target heap

OUT – ppsDstMemInfo – returned memory information

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

```
PVRSRV_ERROR PVRSRVUnmapDeviceMemory (PVRSRV_DEV_DATA *psDevData,
       PVRSRV_CLIENT_MEM_INFO *psMemInfo)
```

This function unmaps an allocation from a heap previously mapped to by PVRSRVMapDeviceMemoryKM

IN – psDevData – device data

IN – psMemInfo – memory information structure

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

```
PVRSRV_ERROR PVRSRVMapExtMemory (PVRSRV_DEV_DATA *psDevData,
       PVRSRV_CLIENT_MEM_INFO *psMemInfo,
       IMG_SYS_PHYADDR *psSysPAddr,
       IMG_UINT32 ui32Flags)
```

This function maps a list of physical pages to (previously reserved) device virtual memory

IN – psDevData – device data

IN – psMemInfo – memory information structure

IN – psSysPAddr – list of system physical page addresses

IN - ui32Flags - flags

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

```
PVRSRV_ERROR PVRSRVUnmapExtMemory (PVRSRV_DEV_DATA *psDevData,
       PVRSRV_CLIENT_MEM_INFO *psMemInfo,
       IMG_UINT32 ui32Flags)
```

This function unmaps a list of physical pages from (previously reserved) device virtual memory

IN – psDevData – device data

IN – psMemInfo – memory information structure

IN - ui32Flags - flags

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

```
PVRSRV_ERROR PVRSRVMapDeviceClassMemory (PVRSRV_DEV_DATA *psDevData,
       IMG_HANDLE hDevMemContext,
       IMG_HANDLE hDeviceClassBuffer,
       PVRSRV_CLIENT_MEM_INFO *psMemInfo)
```

This function maps a list of physical pages to (previously reserved) device virtual memory

IN - psDevData – device data

IN – hDevMemContext – device memory context

IN – hDeviceClassBuffer – handle for 3rd party display buffer to be mapped

IN – psMemInfo – memory structure describing the device virtual address to map RAM to

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

```
PVRSRV_ERROR PVRSRVUnmapDeviceClassMemory (PVRSRV_DEV_DATA *psDevData,
       PVRSRV_CLIENT_MEM_INFO *psMemInfo)
```

This function unmaps a 3rd party display buffer from a (previously reserved) device virtual memory range

IN - psDevData – device data

IN – psMemInfo – memory information structure

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

```
PVRSRV_ERROR IMG_CALLCONV PVRSRVWrapExtMemory(
       PVRSRV_DEV_DATA *psDevData,
       IMG_HANDLE
       hDevMemContext,
       IMG_UINT32 ui32ByteSize,
       IMG_UINT32 ui32PageOffset,
       IMG_BOOL bPhysContig,
       IMG_SYS_PHYADDR *psSysAddr,
       PVRSRV_CLIENT_MEM_INFO **ppsMemInfo);
```

This function allocates a device virtual address and maps a list of physical pages to the device virtual memory

IN – psDevData – device data structure

IN – hDevMemContext – device memory context

IN – ui32ByteSize – bytes to map

IN – ui32PageOffset – offset into the first page referring to the start of the allocation

IN – bPhysContig – is the memory is physically contiguous. Note: currently only support physically contiguous memory

IN – psSysAddr – physical page list

OUT – ppsMemInfo – memory information structure

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

```
PVRSRV_ERROR IMG_CALLCONV PVRSRVUnwrapExtMemory(
       PVRSRV_DEV_DATA *psDevData, PVRSRV_KERNEL_MEM_INFO *psMemInfo);
```

This function unmaps a list of physical pages from device virtual memory and then frees device virtual address range

IN – psDevData – device data structure

IN – psMemInfo – memory information structure

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

6.5. Services Client Support API

```
PVRSRV_ERROR IMG_CALLCONV PVRSRVClientEvent (IMG_CONST PVRSRV_CLIENT_EVENT eEvent,
PVRSRV_DEV_DATA *psDevData, IMG_PVOID pvData)
```

This function handles events in client drivers which are unrecoverable from within the client driver itself and require support from Services. An example of such an event is a timeout after work submitted to the HW has failed to complete successfully. The action performed is typically a dump of debugging information and/or an attempt to recover the device.

IN – eEvent – event type (see `services.h` for allowed values.)

IN – psDevData – device data which is used to switch the action taken according to the HW device type installed.

IN – pvData – (Optional.) Pointer to client-side data or IMG_NULL.

RETURN

- Success – PVRSRV_OK
- Failure – PVRSRV_ error code

6.6. Miscellaneous Services Information

```
PVRSRV_ERROR IMG_CALLCONV PVRSRVGetMiscInfo (
       PVRSRV_CONNECTION *psConnection, PVRSRV_MISC_INFO *psMiscInfo)
```

This function returns miscellaneous information from services. The following flags are available in psMiscInfo->

`PVRSRV_MISC_INFO_TIMER_PRESENT` – returns SOC timer registers

`PVRSRV_MISC_INFO_CLOCKGATE_PRESENT` – returns SOC clockgating registers

`PVRSRV_MISC_INFO_MEMSTATS_PRESENT` – returns memory statistics (requires a sufficient buffer to be allocated and passed via psMiscInfo->pszMemoryStr.)

`PVRSRV_MISC_INFO_GLOBALEVENTOBJECT_PRESENT` – returns a global event object

`PVRSRV_MISC_INFO_DDKVERSION_PRESENT` – returns a string containing the DDK version (requires a sufficient buffer to be allocated and passed via psMiscInfo->pszMemoryStr.)

PVRSRV_MISC_INFO_CPUCACHEFLUSH_PRESENT – provides CPU cache flush controls

Multiple options can be passed together except `PVRSRV_MISC_INFO_MEMSTATS_PRESENT` and `PVRSRV_MISC_INFO_DDKVERSION_PRESENT` cannot be combined.

IN – psConnection – connection information

OUT – psMiscInfo – miscellaneous information returned by services

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

```
PVRSRV_ERROR IMG_CALLCONV PVRSRVReleaseMiscInfo (
       PVRSRV_CONNECTION *psConnection, PVRSRV_MISC_INFO *psMiscInfo)
```

This function releases any resources and mappings associated with PVRSRVGetMiscInfo

IN – psConnection – connection information

IN – psMiscInfo – miscellaneous information returned by services

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

7. Client Services Internal Interfaces

7.1. Operating System Interface

See the Services Porting Guide for API details

8. Memory Allocations

This section describes the main memory allocations made by Services. These allocations are in two main forms: device memory and host memory.

8.1. Device Memory Allocations

Device memory refers to memory that is allocated and mapped to Services managed devices, i.e. memory that is addressable by devices. The default sizes are approximate and subject to revision of the driver. The heap IDs correspond to those in the SGX API (kernel module) header file.

Allocation	Storage for	Default size	Heap	Controlled by
Render context	SGX render context data	Sizeof(SGXMKIF_HWRENDERCONTEXT)	SGX_KERNEL_VIDEO_DATA_HEAP_ID	SGXCreateRenderContext
CCB buffer	SGX client command buffer	64Kb	SGX_KERNEL_VIDEO_DATA_HEAP_ID	SGXCreateRenderContext
TA/3D sync object	SGX sync object	Sizeof(PVRSRV_SYNC_DATA)	SGX_SYNCINFO_HEAP_ID	SGXCreateRenderContext
Scratch primitive block	SGX primitive data	64b	SGX_3DPARAMETERS_HEAP_ID	SGXCreateRenderContext
State restore	SGX state data	52b	SGX_3DPARAMETERS_HEAP_ID	SGXCreateRenderContext
Visibility test result buffer	SGX visibility results buffer	User	SGX_SYNCINFO_HEAP_ID	SGXCreateRenderContext
VDM context switch USE program	SGX USSE code fragment	80b	SGX_KERNEL_CODE_HEAP_ID	CreateContextSwitchStateUpdate
VDM context switch PDS program	SGX PDS code fragment	80b	SGX_KERNEL_CODE_HEAP_ID	CreateContextSwitchStateUpdate
Parameter Buffer	SGX internal parameter data (shared)	4Mb	SGX_3DPARAMETERS_HEAP_ID	CreateSharedPB
DPM page table	SGX internal PT	256Kb	SGX_3DPARAMETERS_HEAP_ID	CreateSharedPB
Microkernel buffer descriptor	SGX internal data description	Sizeof(SGXMKIF_HWPBDESC)	SGX_KERNEL_VIDEO_DATA_HEAP_ID	CreateSharedPB
HW parameter buffer block	SGX internal PB block	Sizeof(SGXMKIF_HWPBBLOCK)	SGX_KERNEL_VIDEO_DATA_HEAP_ID	CreatePerContextPB
Parameter Buffer	SGX internal parameter data	4Mb	SGX_3DPARAMETERS_HEAP_ID	CreatePerContextPB
DPM page table	SGX internal PT	256Kb	SGX_3DPARAMETERS_HEAP_ID	CreatePerContextPB
Microkernel buffer descriptor	SGX internal data description	Sizeof(SGXMKIF_HWPBDESC)	SGX_3DPARAMETERS_HEAP_ID	CreatePerContextPB
HW RT data set	SGX render target dataset common data	Sizeof(SGXMKIF_HWRTDATASET)	SGX_KERNEL_VIDEO_DATA_HEAP_ID	SetupRTDataset
Tail pointers	SGX tail pointer data	render target tile count x 4bytes	SGX_TADATA_HEAP_ID	SetupRTDataset
Context control	SGX DPM control	64b	SGX_TADATA_HEAP_ID	SetupRTDataset
Context OTPM	SGX OTPM internal data	(((NumMT(4 or 16) * 4) + 15) & ~15)	SGX_TADATA_HEAP_ID	SetupRTDataset
Special objects	Special objects	512b	SGX_3DPARAMETERS_HEAP_ID	SetupRTDataset
Region headers	Region headers	render target tile count x 12bytes	SGX_TADATA_HEAP_ID	SetupRTDataset
Context state	SGX PB internal state	approx 402 bytes per core	SGX_TADATA_HEAP_ID	SetupRTDataset
VDM snapshot buffer	SGX VDM state snapshot	496b	SGX_KERNEL_VIDEO_DATA_HEAP_ID	SetupRTDataset
PDS state	PDS state storage	4Kb per PDS pipe	SGX_KERNEL_VIDEO_DATA_HEAP_ID	SetupRTDataset
ZLS tile buffer for context switch	SGX depth-test data	ISP_xsize x ISP_ysize x 5 bytes	SGX_KERNEL_VIDEO_DATA_HEAP_ID	SetupRTDataset
Tile LUT	SGX lookup table	4 bytes per tile	SGX_KERNEL_VIDEO_DATA_HEAP_ID	SetupRTDataset
Macrotile LUT	SGX lookup table	4 bytes per macrotile	SGX_KERNEL_VIDEO_DAT	SetupRTDataset

Allocation	Storage for	Default size	Heap	Controlled by
			A_HEAP_ID	
RTData status	SGX render target	4 bytes per render target	SGX_KERNEL_VIDEO_DATA_HEAP_ID	SetupRTDataset
Dev sync list	SGX device sync data	Sizeof(SGXMKIF_HWDEVICE_SYNC_LIST)	SGX_KERNEL_VIDEO_DATA_HEAP_ID	SetupRTDataset
Access resource	SGX device sync data	Sizeof(PVRSRV_RESOURCE)	SGX_SYNCINFO_HEAP_ID	SetupRTDataset
Render details	SGX render details	Sizeof(SGXMKIF_HWRENDERDETAILS	SGX_KERNEL_VIDEO_DATA_HEAP_ID	CreateRenderDetails
Access resource	SGX device sync data	Sizeof(PVRSRV_RESOURCE)	SGX_SYNCINFO_HEAP_ID	CreateRenderDetails
Render details	SGX render details	8 bytes per render target	SGX_KERNEL_VIDEO_DATA_HEAP_ID	CreateRenderDetails
Primary PDS update list	SGX PDS updates	4 bytes per render target	SGX_KERNEL_VIDEO_DATA_HEAP_ID	CreateRenderDetails
Staging buffer	SGX TQ buffer	(user!=0) ? max(user, SGXTQ_STAGINGBUFFER_MIN_SIZE) : 0	SGX_2D_HEAP_ID or SGX_GENERAL_HEAP_ID	SGXCreateTransferContext
Staging ROF	SGX TQ ROF	4b	SGX_SYNCINFO_HEAP_ID	SGXCreateTransferContext
TA/TQ dep sync object	SGX sync data	sizeof(PVRSRV_SYNC_DATA)	SGX_SYNCINFO_HEAP_ID	SGXCreateTransferContext
TQ CCB	SGX client command buffer for transfers	64Kb	SGX_KERNEL_VIDEO_DATA_HEAP_ID	SGXCreateTransferContext
3D/TQ dep sync object	SGX sync object	sizeof(PVRSRV_SYNC_DATA)	SGX_SYNCINFO_HEAP_ID	SGXCreateTransferContext
HW transfer context	SGX transfer context	Sizeof(SGXMKIF_HWTRANSFERCONTEXT)	SGX_KERNEL_VIDEO_DATA_HEAP_ID	SGXCreateTransferContext
PDS state context switch	PDS state store	4Kb	SGX_KERNEL_VIDEO_DATA_HEAP_ID	SGXCreateTransferContext
2D CCB	SGX client command buffer for 2D operations	64Kb	SGX_KERNEL_VIDEO_DATA_HEAP_ID	SGXCreateTransferContext
HW 2D context	SGX BLT context	Sizeof(SGXMKIF_HW2DCONTEXT)	SGX_KERNEL_VIDEO_DATA_HEAP_ID	SGXCreateTransferContext
Secondary transfer context	SGX transfer context	Sizeof(SGXMKIF_HW2DCONTEXT)	SGX_KERNEL_VIDEO_DATA_HEAP_ID	SGXCreateTransferContext
Secondary transfer CCB	SGX client command buffer for transfers	64Kb	SGX_KERNEL_VIDEO_DATA_HEAP_ID	SGXCreateTransferContext
Secondary 2D context	SGX BLT context	Sizeof(SGXMKIF_HW2DCONTEXT)	SGX_KERNEL_VIDEO_DATA_HEAP_ID	SGXCreateTransferContext
Secondary 2D CCB	SGX client command buffer for 2D operations	64Kb	SGX_KERNEL_VIDEO_DATA_HEAP_ID	SGXCreateTransferContext
USE code for blit	SGX USSE code fragment	depends on blit	SGX_PIXELSHADER_HEAP_ID	SGXCreateTransferContext
Primary PDS	SGX PDS code fragment	depends on blit	SGX_PDSPIXEL_CODEDATA_HEAP_ID	SGXCreateTransferContext
Secondary PDS (workaround)	SGX PDS code fragment	80b	SGX_PDSPIXEL_CODEDATA_HEAP_ID	SGXCreateTransferContext
Secondary PDS	SGX PDS code fragment	depends on blit	SGX_PDSPIXEL_CODEDATA_HEAP_ID	SGXCreateTransferContext
EOT USE event	SGX USSE code fragment	1056b	SGX_PIXELSHADER_HEAP_ID	SGXCreateTransferContext
EOT program for	SGX USSE code	4136b	SGX_PIXELSHADER_HEAP	SGXCreateTransferContext

Allocation	Storage for	Default size	Heap	Controlled by
subpixel twiddling	fragment		_ID	
EOT USE code setup	SGX USSE code fragment	max of (11+STATE_PBE_DWORDS)*8	SGX_PIXELSHADER_HEAP_ID	SGXCreateTransferContext
Control object for fast renders	Fast blit control stream	(11+6*num layers)*4b	SGX_GENERAL_HEAP_ID	SGXCreateTransferContext
ISP control stream skeleton	ISP control stream	112b	SGX_GENERAL_HEAP_ID	SGXCreateTransferContext
EOR handler	SGX PDS code fragment	24b	SGX_PDSPIXEL_CODEDATA_HEAP_ID	SGXCreateTransferContext
PDS Pixel event program	SGX PDS code fragment	Sizeof(PIXEVENT_PROG_SIZE)	SGX_PDSPIXEL_CODEDATA_HEAP_ID	SGXCreateTransferContext
TA state restore program (USSE)	SGX USSE code fragment	200b	SGX_KERNEL_CODE_HEAP_ID	LoadTAStateRestoreProgram
TA state restore program (PDS)	SGX PDS code fragment	20b	SGX_KERNEL_CODE_HEAP_ID	LoadTAStateRestoreProgram
Microkernel	SGX USSE microkernel program	up to 5x 32Kb	SGX_KERNEL_CODE_HEAP_ID	LoaduKernelProgram
Microkernel PDS	SGX PDS microkernel program	68 bytes	SGX_KERNEL_CODE_HEAP_ID	LoaduKernel
Loopback PDS	SGX microkernel loopback program	142 bytes	SGX_KERNEL_CODE_HEAP_ID	LoaduKernel
Kernel CCB	SGX kernel command buffer	sizeof(PVRSRV_SGX_KERNEL_CCB)	SGX_KERNEL_VIDEO_DATA_HEAP_ID	SetupuKernel
Kernel CCB control	SGX kernel command buffer control	sizeof(PVRSRV_SGX_CCB_CTL)	SGX_KERNEL_VIDEO_DATA_HEAP_ID	SetupuKernel
SGX kernel event kicker	SGX event kick register	4b	SGX_KERNEL_VIDEO_DATA_HEAP_ID	SetupuKernel
TA/3D control	SGX TA/3D internal control structure	sizeof(SGXMK_TA3D_CTL)	SGX_KERNEL_VIDEO_DATA_HEAP_ID	SetupuKernel
Host control	SGX host control structure	sizeof(SGXMKIF_HOST_CTL)	SGX_KERNEL_VIDEO_DATA_HEAP_ID	SetupuKernel
SGX misc info buffer	SGX device misc info buffer	sizeof(PVRSRV_SGX_MISCINFO_INFO)	SGX_KERNEL_VIDEO_DATA_HEAP_ID	SetupuKernel
Profiling	SGX HW profiling	sizeof(SGXMKIF_HWPROFILING)	SGX_KERNEL_VIDEO_DATA_HEAP_ID	SetupuKernel
Perf buffer	SGX HW perf counter buffer	sizeof(SGXMKIF_HWPERF_CB)	SGX_KERNEL_VIDEO_DATA_HEAP_ID	SetupuKernel
Temp region headers	SGX ISP region headers	((EURASIA_RENDERSIZE_MAXX / EURASIA_ISPREGION_SIZEX) * (EURASIA_RENDERSIZE_MAXY / EURASIA_ISPREGION_SIZEY) *EURASIA_REGIONHEADER_SIZE) / 4	SGX_KERNEL_VIDEO_DATA_HEAP_ID	SetupuKernel
Temporary DPM state table	SGX DPM parameter state table	Sizeof(EURASIA_PARAM_MANAGE_STATE_SIZE)	SGX_KERNEL_VIDEO_DATA_HEAP_ID	SetupuKernel

Note: For each allocation of device-addressable memory, an additional allocation of size `PVRSRV_KERNEL_MEMINFO` is made to hold the CPU (host) and device virtual addresses of the buffer. In most cases a sync object and resource manager object are also allocated.

8.2. Host Memory Allocations (Client services module)

Host memory refers to memory that is allocated for Services data and control structures

Allocation	Storage for	Default size	Controlled by
3D signature registers	SGX register data	(num3Dregs*numMPcores)*4 bytes	PDump3DSignatureRegisters
TA signature registers	SGX register data	(numTAregs*numMPcores)*4 bytes	PDumpTASignatureRegisters
SGX host render context	SGX host render context	sizeof(SGX_RENDERCONTEXT)	SGXCreateRenderContext
SGX host CCB	SGX host command buffer	sizeof(SGX_CLIENT_CCB)	SGXCreateRenderContext
SGX host PB data	SGX host parameter buffer data	sizeof(SGX_CLIENTPBDESC)	SGXCreateRenderContext
SGX host PB	SGX host parameter buffer	sizeof(SGX_PBDESC)	CreatePerContextPB
PB block	SGX host PB block	sizeof(SGX_PBBLOCK)	CreatePerContextPB
SGX host RT data	SGX host render target data	sizeof(SGX_RTDATASET)	SGXAddRenderTarget
SGX host device sync data	SGX host device sync list	sizeof(SGX_DEVICE_SYNC_LIST)	CreateDeviceSyncList
SGX host render details	SGX host render details	sizeof(SGX_RENDERDETAILS)	CreateRenderDetails
SGX host transfer context	SGX host transfer context	sizeof(SGXTQ_CLIENT_TRANSFER_CONTEXT)	SGXCreateTransferContext
SGX TQ host CCB	SGX host transfer queue command buffer	sizeof(SGX_CLIENT_CCB)	SGXCreateTransferContext
SGX 2D host CCB	SGX host 2D command buffer	sizeof(SGX_CLIENT_CCB)	SGXCreateTransferContext
SGX HW Background objects	SGX internal background objects	sizeof(IMG_DEV_VIRTADDR) * SGXTQ_NUM_HWBGOBJS	SGXCreateTransferContext
SGX HW Background objects	SGX internal background objects	sizeof(PVRSRV_CLIENT_MEM_INFO*) * SGXTQ_NUM_HWBGOBJS	SGXCreateTransferContext
SGX pixel event info	SGX transfer queue pixel event client info	sizeof(SGXTQ_CLIENT_TRANSFER_PIXEVENT) * SGXTQ_NUM_PDSPIXELEVENTS	SGXCreateTransferContext
USSE program frags	USSE transfer queue program addresses	sizeof(IMG_DEV_VIRTADDR) * SGXTQ_NUM_USEFRAGS	SGXCreateTransferContext
USSE program frags	USSE transfer context program meminfos	sizeof(PVRSRV_CLIENT_MEM_INFO *) * SGXTQ_NUM_USEFRAGS	SGXCreateTransferContext
Temp register numbers	SGX USSE temporary register numbers	sizeof(IMG_UINT32) * SGXTQ_NUM_USEFRAGS	SGXCreateTransferContext
PA register numbers	SGX USSE PA register numbers	sizeof(IMG_UINT32) * SGXTQ_NUM_USEFRAGS	SGXCreateTransferContext
PDS primary DevVAddrs	PDS transfer queue program addresses	sizeof(IMG_DEV_VIRTADDR) * SGXTQ_NUM_PDSPRIMFRAGS	SGXCreateTransferContext
PDS primary data segments	PDS transfer queue primary program data	sizeof(IMG_UINT32) * SGXTQ_NUM_PDSPRIMFRAGS	SGXCreateTransferContext
PDS primary frags	PDS transfer context program meminfos	sizeof(PVRSRV_CLIENT_MEM_INFO *) * SGXTQ_NUM_PDSPRIMFRAGS	SGXCreateTransferContext
PDS sec frags	PDS transfer queue secondary program addresses	sizeof(IMG_DEV_VIRTADDR) * SGXTQ_NUM_PDSSECFRAGS	SGXCreateTransferContext
PDS sec frag sizes	PDS transfer queue secondary program sizes	sizeof(IMG_UINT32) * SGXTQ_NUM_PDSSECFRAG	SGXCreateTransferContext

Allocation	Storage for	Default size	Controlled by
		S	
PDS sec frag sizes	PDS transfer queue secondary program sizes	sizeof(IMG_UINT32) * SGXTQ_NUM_PDSSECFRAGS	SGXCreateTransferContext
PDS sec frags	PDS transfer context secondary program meminfos	sizeof(PVRSRV_CLIENT_MEM_INFO *) * SGXTQ_NUM_PDSSECFRAGS	SGXCreateTransferContext

8.3. Host Memory Allocations (Kernel services module)

Host memory refers to memory that is allocated for Services data and control structures

Allocation	Storage for	Default size	Heap	Controlled by
Display class pointer	Pointer to PVR display class info	sizeof(PVRSRV_DISPLAYCLASS_INFO*)	OS_Pageable	PVRSRVRegisterDCDeviceKM
Device node	PVR device node	sizeof(PVRSRV_DEVICE_NODE)	OS_Pageable	PVRSRVRegisterDCDeviceKM
Buffer class pointer	Pointer to PVR buffer class info	sizeof(PVRSRV_BUFFERCLASS_INFO*)	OS_Pageable	PVRSRVRegisterBCDeviceKM
Services to third-party BC API jump table	PVR buffer class API jump table	sizeof(PVRSRV_BC_SRV2BUFFER_KMJTABLE)	OS_Pageable	PVRSRVRegisterBCDeviceKM
Third-party BC device node	PVR buffer class device	sizeof(PVRSRV_DEVICE_NODE)	OS_Pageable	PVRSRVRegisterBCDeviceKM
Per context DC info	PVR per context display class info plus sync object data	sizeof(PVRSRV_DISPLAYCLASS_PERCONTEXT_INFO) plus sync object data	OS_Pageable	PVRSRVOpenDCDeviceKM
Display class swap chain	PVR display class swap chain	sizeof(PVRSRV_DC_SWAPCHAIN)	OS_Pageable	PVRSRVCreateDCSwapChainKM
Per context BC info	PVR per context buffer class info	sizeof(PVRSRV_BUFFERCLASS_PERCONTEXT_INFO)	OS_Pageable	PVRSRVOpenBCDeviceKM
Buffer class data	PVR buffer class internal data	sizeof(PVRSRV_BC_BUFFER) per buffer	OS_Pageable	PVRSRVOpenBCDeviceKM
Mem info block	PVR mem info block	sizeof(PVRSRV_KERNEL_MEM_INFO)	OS_Non_Pageable	PVRSRVReserveDeviceVirtualMemKM
Page tables	PVR device virtual mem page tables	sizeof(IMG_SYS_PHYADDR) per page	OS_Pageable	PVRSRVWrapExtMemoryKM
Mem info block	PVR mem info block and PVR sync object	sizeof(PVRSRV_KERNEL_MEM_INFO) plus sync object data	OS_Pageable	PVRSRVWrapExtMemoryKM
Page tables	PVR device virtual mem page tables	sizeof(IMG_SYS_PHYADDR) per page	OS_Pageable	PVRSRVMapDeviceMemoryKM
Resman info	PVR resource manager data	sizeof(RESMAN_MAP_DEVICE_MEM_DATA)	OS_Pageable	PVRSRVMapDeviceMemoryKM
Reallocate mem area	Reallocated buffer	user	OS_Pageable	ReallocMem
Per process data area	PVR per process data	sizeof(PVRSRV_PER_PROCESS_DATA)	OS_Non_Pageable	PVRSRVPerProcessDataConnect
Device power management data	PVR device power manager data	sizeof(PVRSRV_POWER_DEV)	OS_Pageable	PVRSRVRegisterPowerDevice
Event object	PVR event object	sizeof(PVRSRV_EVENTOBJECT)	Pageable_Select	PVRSRVInit
Device node	PVR device	sizeof(PVRSRV_DEVICE_NODE)	OS_Non_Pageable	PVRSRVRegisterDevice
Internal queue info	PVR internal command queue data	sizeof(PVRSRV_QUEUE_INFO)	Non_OS_Pageable	PVRSRVCreateCommandQueueKM
Command queue	PVR internal command queue	user (rounded to power of 2)	Non_OS_Pageable	PVRSRVCreateCommandQueueKM
Internal DC command queue data	Pointer to PVR queued command	sizeof(PFN_CMD_PROC) per command	OS_Pageable	PVRSRVRegisterCmdProcListKM
Command complete data pointer	Pointer to PVR command complete data	sizeof(COMMAND_COMPLETE_DATA*) per command	OS_Pageable	PVRSRVRegisterCmdProcListKM
Sync object on command complete	PVR sync objects	Sizeof(PVRSRV_SYNC_OBJECT) * max sync objects for src and dest surfaces	Non_OS_Pageable	PVRSRVRegisterCmdProcListKM

Allocation	Storage for	Default size	Heap	Controlled by
PT info	PVR MMU page table data	sizeof(MMU_PT_INFO) per page table	OS_Pageable	_DeferredAllocPagetables
MMU context	PVR MMU context	sizeof(MMU_CONTEXT)	OS_Pageable	MMU_Initialise
MMU device	PVR MMU heap	sizeof(MMU_HEAP)	OS_Pageable	MMU_Create
PT info	PVR MMU page table data	sizeof(MMU_PT_INFO) per page table	OS_Pageable	_DeferredAllocPagetables
MMU context	PVR MMU context	sizeof(MMU_CONTEXT)	OS_Pageable	MMU_Initialise
MMU device	PVR MMU heap	sizeof(MMU_HEAP)	OS_Pageable	MMU_Create
PB mem object	Pointer to PVR mem info block	sizeof(PVRSRV_KERNEL_MEM_INFO*) per subkernel meminfo	OS_Pageable	SGXFindSharedPBDescKM
Shared PB	PVR internal shared PB description	sizeof(PVRSRV_STUB_PBDESC)	Non_OS_Pageable	SGXAddSharedPBDescKM
PB mem object	Pointer to PVR mem info block	sizeof(PVRSRV_KERNEL_MEM_INFO*) per subkernel meminfo	Non_OS_Pageable	SGXAddSharedPBDescKM
Internal CCB info	SGX internal command buffer data	sizeof(PVRSRV_SGX_CCB_INFO)	OS_Pageable	InitDevInfo
Device control block	SGX device info	sizeof(PVRSRV_SGXDEV_INFO)	Non_OS_Pageable	DevInitSGXPart1
Device memory heaps	SGX device memory heap info	sizeof(DEVICE_MEMORY_HEAP_INFO) per heap	OS_Pageable	SGXRegisterDevice
Render context cleanup	SGX internal render context clean-up data	sizeof(SGX_HW_RENDER_CONTEXT_CLEANUP)	OS_Pageable	SGXRegisterHWRenderContextKM
Transfer context cleanup	SGX internal transfer context clean-up data	sizeof(SGX_HW_TRANSFER_CONTEXT_CLEANUP)	OS_Pageable	SGXRegisterHWTransferContextKM
2D cleanup	SGX internal 2D context clean-up data	sizeof(SGX_HW_2D_CONTEXT_CLEANUP)	OS_Pageable	SGXRegisterHW2DContextKM

Appendix A. Application, Driver, Microkernel and Hardware Interactions

This appendix describes some of the complex interactions between the various software and hardware components:

- Application and client driver, e.g. OGLES application and driver
- Services client library
- Kernel Services Driver
- SGX USSE Microkernel
- SGX hardware (TA, 3D, USSE)
- SGX and Display LISRs
- Services MISR

UML sequence diagrams are used to illustrate the component interactions for a number of use case examples.

A.1. SGX Render and Display Flip use case

This example use case is of an application submitting two scenes for rendering, each of which is to be flipped on the display hardware and presented for at least one blanking period.

Figure 6 describes how the driver and microkernel manage the asynchronous command processing as well as handling display and SGX hardware events.

Figure 7 describes the same use case sequence as Figure 6 SGX Render and Display Flip use case but includes SPM events.

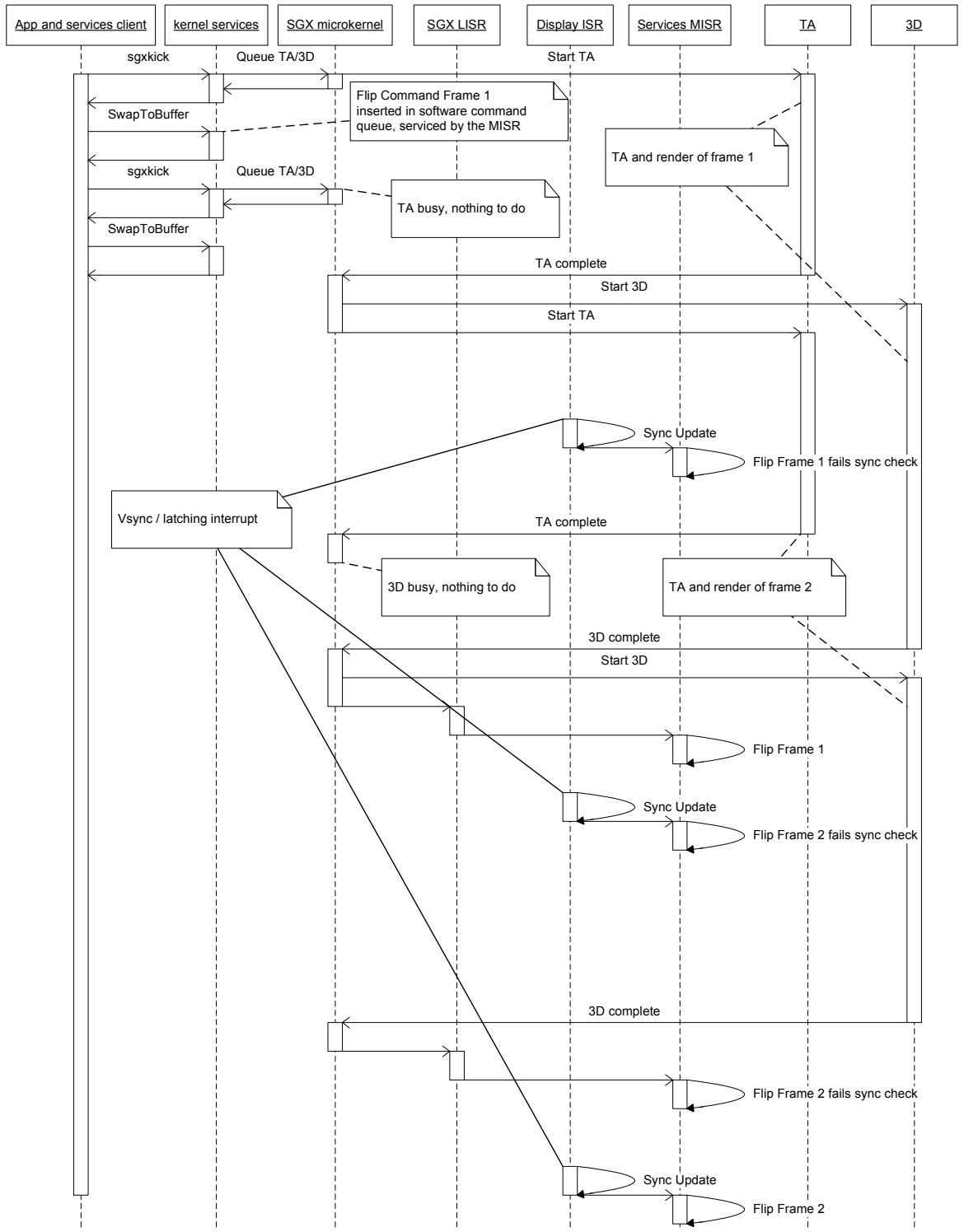

Figure 6 SGX Render and Display Flip use case

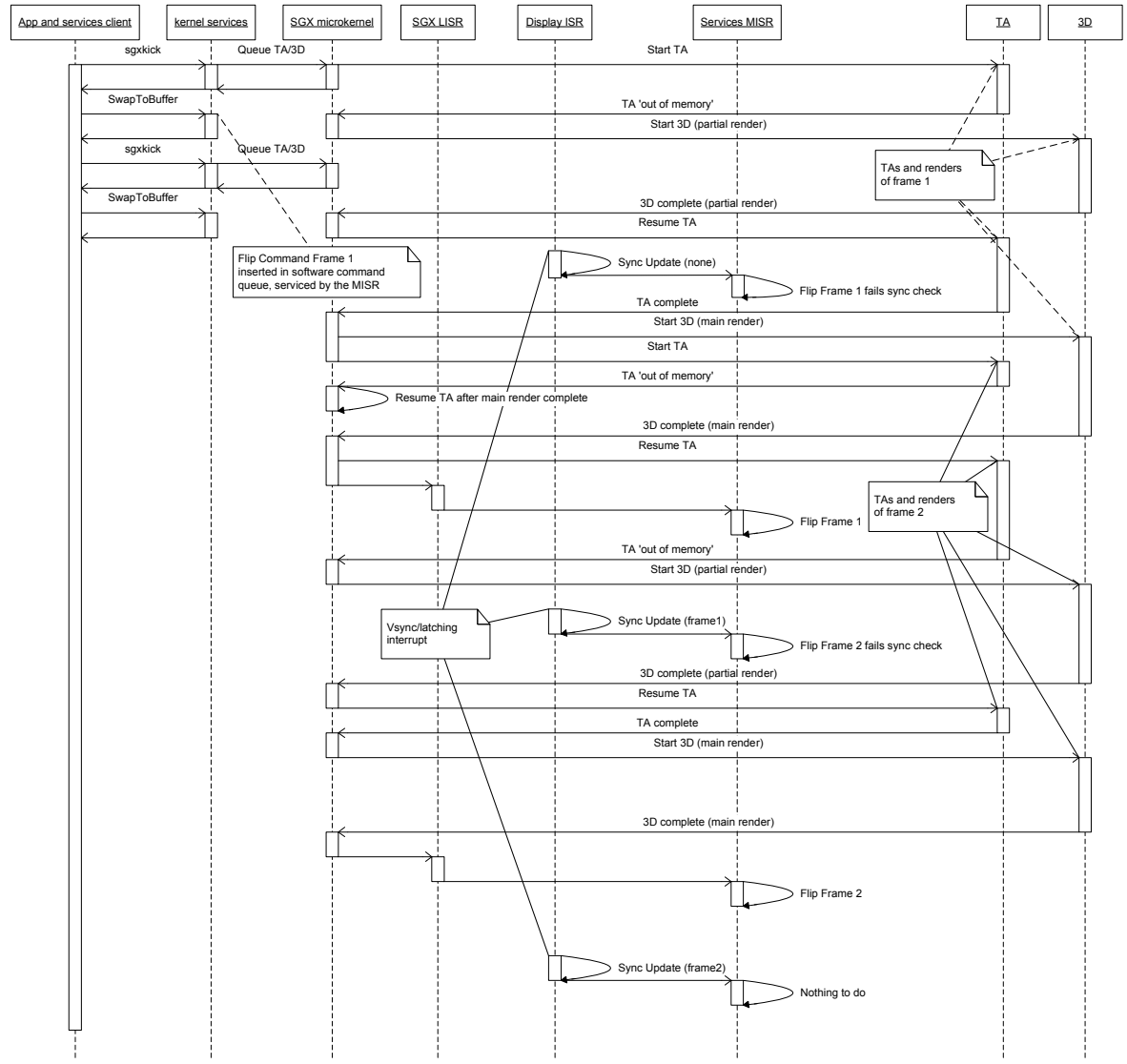

Figure 7 SGX Render and Display Flip use case with SPM

Appendix B. Parameter Buffer Grow / Shrink Support

This appendix provides an overview of how parameter buffer (PB) grow / shrink works and can be controlled.

B.1. Controllable options

There are 2 variables which control parameter buffer grow / shrink. These are the initial size and the maximum size. Both of these variables are arguments to the `SGXCreateRenderContext` API via the `PSGX_CREATERENDERCONTEXT` structure.

`ui32PBSize`	This indicates the initial requested parameter buffer size in bytes.
`ui32PBSizeLimit`	This is a new variable which when non-zero indicates the maximum size the parameter buffer should be allowed to grow to.

B.2. Methodology

B.2.1. Grow

The method of growing a parameter buffer is based around the SGX out of memory event. This event indicates that the current parameter buffer size is not large enough to render the current geometry load without at best serialising TA and 3D activity or at worse entering a SPM partial render cycle. The act of SGX generating this event, results in the uKernel setting a flag in the `SGXMKIF_HWRTDATA` for the render target currently being TA'd.

This flag is then check by the host driver in the `SGXKickTA` function on every kick. When the host driver detects this flag is set it calls the `GrowPerContextPB` function. This function is responsible for checking that the maximum parameter buffer size limit is not exceeded and if not, the setup of a new parameter buffer block. As a result of the command buffering between the HW and the host driver it is possible for the driver to inject more blocks than are required to avoid hitting SPM. However, as soon as the driver detects that no more SPM events have been generated it will start shrinking the parameter buffer size down to the minimum required to avoid SPM.

The new threshold and size variables along with the details of the parameter block linked list update are then inserted into the `SGXMKIF_HWPBDESC_UPDATE` structure of the next TA command to be submitted. When the uKernel comes to process the command it will link in the new parameter buffer block and update the DPM thresholds to reflect the increase available memory.

B.2.2. Shrink

In order to shrink the parameter buffer, the host srvclient module code tracks whether there are any scenes in progress. This is required as is only safe for the uKernel to remove parameter buffer blocks when the whole of the parameter buffer is free. This is because the parameter buffer becomes fragmented during use.

This is done in `SGXKickTA` by using the `ui32BusySceneCount` member of the `SGX_PBDESC` structure. When the value is 0 it is safe to queue a shrink of the parameter buffer. The driver is only capable of removing 1 block at a time and can never shrink beyond that of the initial size. In order to avoid thrashing on the parameter buffer size, a hysteresis algorithm has been put it place where by the parameter buffer usage (as recorded by the uKernel on out of memory and TA finished events) must remain at least (`SGXPB_SHRINK_TEST_MULTIPLIER * ui32PBGrowBlockSize`) below the current size for more than `SGX_SHRINK_MIN_FRAMES` number of renders.