SGX Services

Porting Guide for Version 4.0

Filename	:	SGX Services Porting Guide for Version 4 0.doc
Version	:	1.1.197 External Issue
Issue Date	:	14 Jun 2011
Author	:	PowerVR

Contents

List of Figures

1. Introduction

This document describes the PowerVR Drivers DDK porting functions in order to support new operating systems and hardware platforms. Within the DDK it is the 'Services' software components that are responsible for abstracting operating system and hardware platform variance as well as device control. This version of the Services API is referred to as 4.0. This generation of Services has been created to cover the SGX generation of hardware. Only Services porting functions need to be ported to support all relevant APIs in a specific platform. This document will cover these functions.

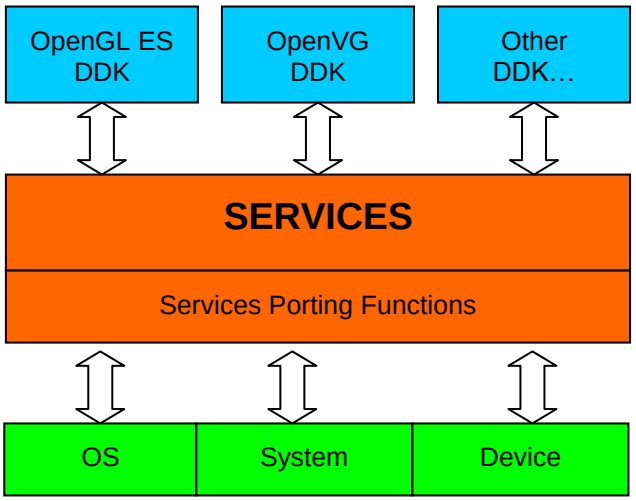


Figure 1 Services Overview

2. Terminology

The following terminology is used in this document:

BSP	Board Support Package. A software package supplied by the developer of the board, adding support for the hardware devices and peripherals found on the board.
DDK	Driver Development Kit. A software package containing driver source code, allowing a specific driver to be built\modified for use on a specific platform.
PVR	PowerVR
UMA	Unified Memory Architecture – graphics device addresses system memory either allocated from the OS or reserved in the system memory
LMA	Local Memory Architecture – graphics device has its own block of memory, separate from system memory.
KM	Kernel Mode
UM	User Mode
HW	Hardware
SW	Software

3. Services

3.1. Services overview

Consumer Services is a generalised software integration framework for SOC based platforms running various Operating Systems.

Consumer Services is designed with flexibility in mind and must support:

Current and future HW platforms

Devices in any combination

System architectures (LMA, UMA) and CPUs (ARM, XScale, SH3/4, x86, etc.)

Operating systems (Linux, Symbian, WinCE, WinMobile, etc.)

Design Objectives:

Reduce time and effort in developing software for new projects

Increase code re-use and stability

Flexibility in porting to new platforms and environments

Minimal performance loss compared to customised solutions

3.2. Services Architecture

Split between ‘Client Services’ and ‘Kernel Services’

Client Services provides the services device and memory management and API to client drivers. Depending on the OS and system environment, comes in two flavours:

- Services Client UM – supports User-mode (UM) client drivers, e.g. OGLES
- Services Client KM – supports Kernel-mode (KM) client drivers, e.g. WinXP PC D3D HAL

Kernel Services is a single module providing device and memory management and services calls from all services ‘clients’.

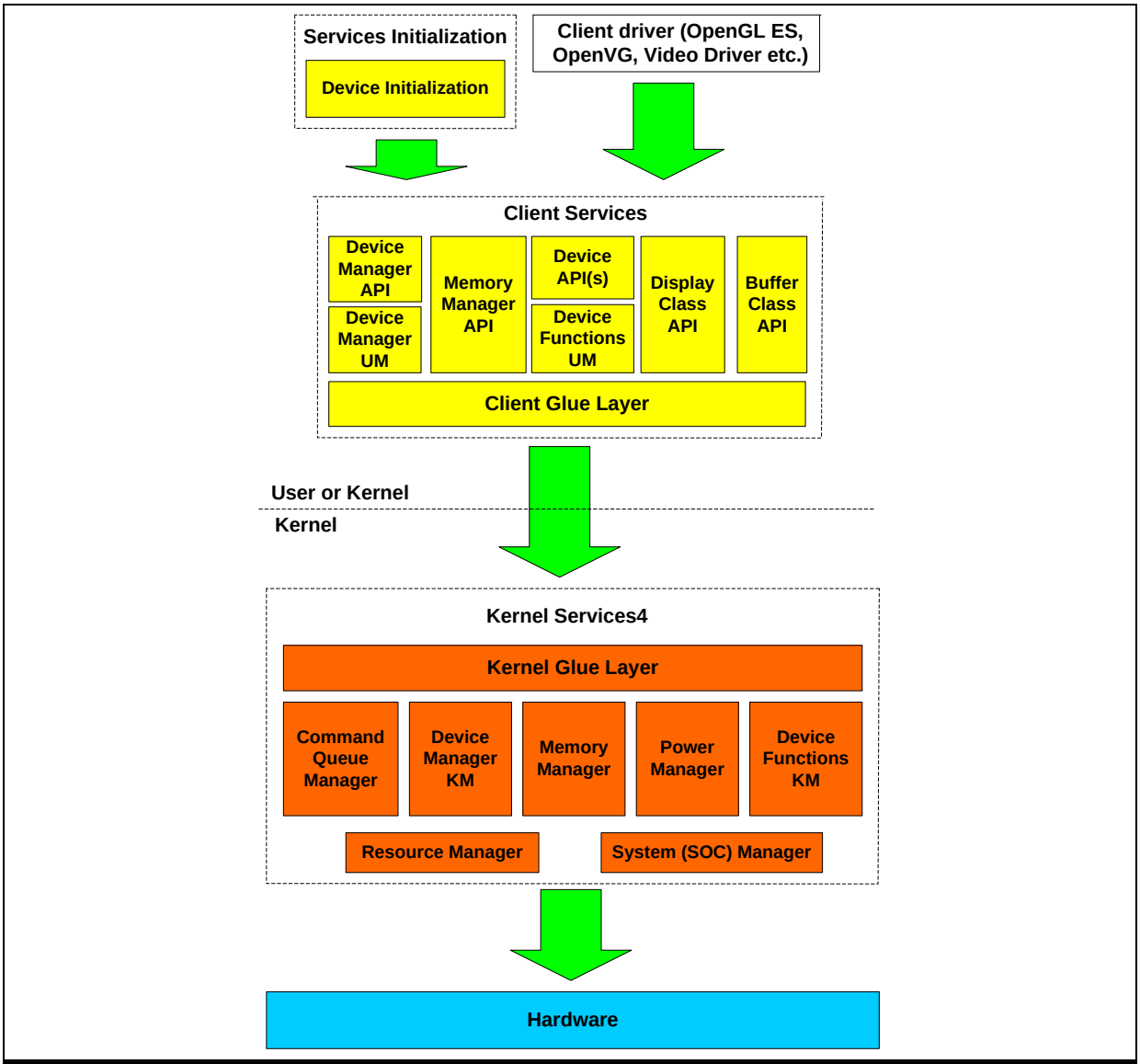


Figure 2 Services Driver Architecture

Note: Refer to 'Services.Software Functional Specification' for detailed descriptions and specifications of component APIs

4. Porting the DDK

4.1. Services porting functions

The ‘Services’ porting functions and are divided into two main sections: Operating System Environment.

section describes the abstraction of the operating system functionality required by Services. It is split into two sub-sections: Client Services and Kernel Services.

Client Services provides the services API for client drivers of ‘Services’. It can be a static library built as part of the client driver or an independent dynamically loaded library. It may exist in user mode or kernel mode depending on the operating system environment.

Note: In the case of no Kernel/User split (e.g. no OS), Client Services acts as a ‘glue layer’ and implements most of the Services’ functionality.

Note: Client drivers always call into kernel services via client services.

Kernel and Client Services both require a fixed set of operating system functionality which is described in the following sections.

The System describes the abstraction a given HW platform (SOC) and, like the Operating System Environment, requires a fixed set of functions to specify the configuration of the underlying HW platform.

To port the PowerVR DDK to a new platform the System functions must be implemented for the platform concerned. As a reference, it is useful for the porting engineer to see sample code for an OS (with kernel and user split) on an example HW platform. Once the port is complete there are Services unit tests which validate if the porting process has been successful. If the unit tests confirm success, it is expected that all the PowerVR drivers will now compile, link and run on the new platform.

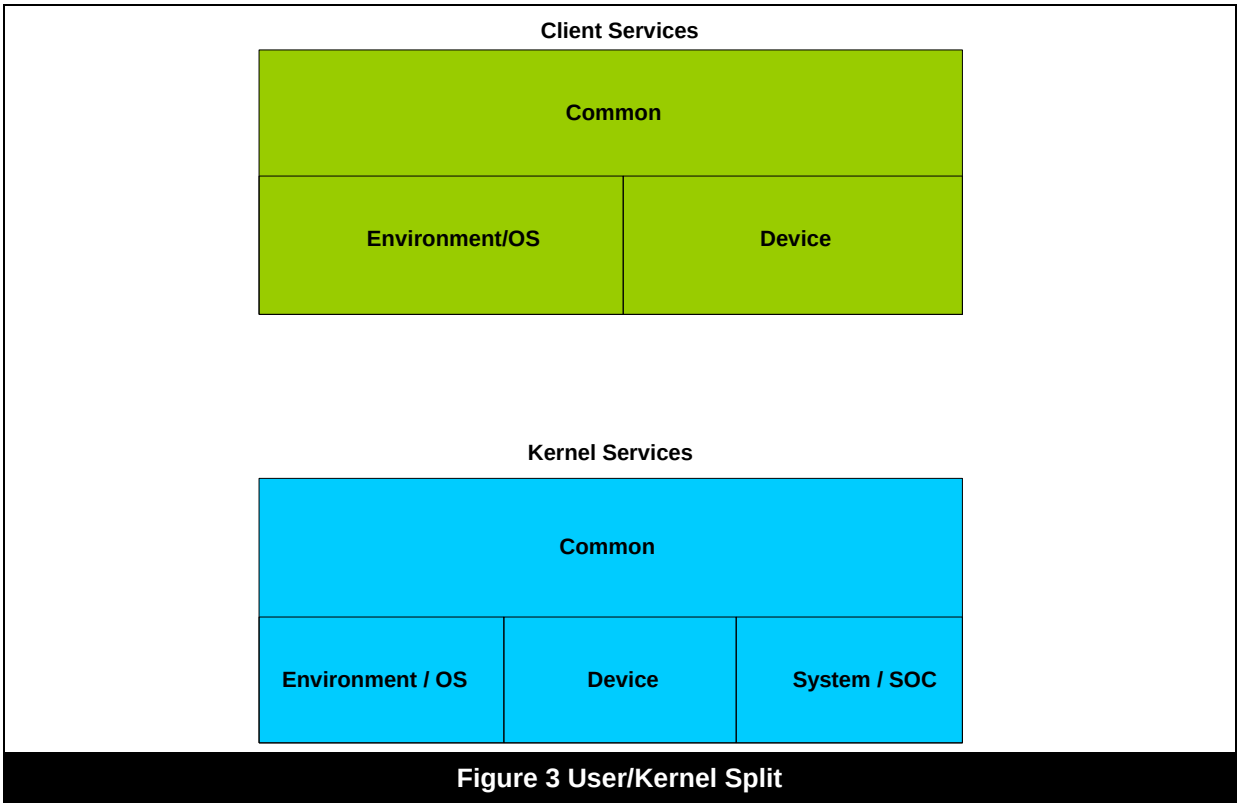


Figure 3 User/Kernel Split

4.2. Adding a new OS

1. Create new OS folders in:
   /services4/srvkm/env
   /services4/srvclient/env

/services4/srvinit/env

2. Use another OS as a template
3. APIs defined in:
   /include4/services.h
   /services4/include/osfunc_client.h
   /services4/srvkm/include/ostfunc.h
4. Be aware that ‘arbitrary’ implementations may have adverse effects on performance/stability
5. Best approach to follow an existing and similar OS implementation

4.3. Adding a new System (SOC)

1. Create new system folder in:
   /system
2. Use existing SOC as a template
3. Common System API defined in:
   /system/include/syscommon.h

5. Operating System Environment Functions

This section describes the operating system abstraction functions that must be implemented in order to port Services to a new operating system. Both Client and Kernel Services have their own OS abstraction APIs

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.

5.1. Client Services

5.1.1. PVRSRVClockus

```
IMG_UINT32 PVRSRVClockus (IMG_VOID)
```

Returns the clock in micro-seconds

RETURN – clock value in microseconds

5.1.2. PVRSRVWaitus

```
IMG_VOID PVRSRVWaitus (IMG_UINT32 ui32Timeus)
```

This function implements a busy wait of specified micro-seconds. This function does not release the current thread quanta.

Note: The accuracy of this function is not essential as it is used in conjunction with PVRSRVClockus which should provide accurate timing (at least in terms of milliseconds).

IN – time to wait in microseconds

5.1.3. PVRSRVMemCopy

```
IMG_VOID PVRSRVMemCopy(IMG_VOID *pvDst, const IMG_VOID *pvSrc, IMG_SIZE_T uiSize)
```

This function is an abstraction of the ANSI memcpy() function.

5.1.4. PVRSRVMemSet

```
IMG_VOID PVRSRVMemSet(IMG_VOID *pvDest, IMG_UINT8 ui8Value, IMG_SIZE_T uiSize)
```

This function is an abstraction of the ANSI memset() function.

5.1.5. PVRSRVGetCurrentProcessID

```
IMG_UINT32 PVRSRVGetCurrentProcessID(IMG_VOID)
```

This function returns a globally unique process identifier for the current executing thread.

RETURN – process ID

5.1.6. PVRSRVAllocUserModeMem

```
IMG_PVOID PVRSRVAllocUserModeMem (IMG_SIZE_T uiSize)
```

This function allocates memory in the address space of the calling application

IN – ui32Size - size bytes to allocate

RETURN

Success – valid pointer to user allocated buffer

Failure – NULL

5.1.7. PVRSRVFreeUserModeMem

```
IMG_VOID PVRSRVFreeUserModeMem (IMG_PVOID pvMem)
```

This function frees memory that was allocated with PVRSRVAllocUserModeMem()

IN – pvMem – pointer to buffer to free

5.1.8. PVRSRVReallocUserModeMem

```
IMG_PVOID PVRSRVReallocUserModeMem(IMG_PVOID pvBase, IMG_SIZE_T uNewSize)
```

This function changes the size of the memory object pointed to by pvBase to the size specified by uNewSize. The contents of the object will remain unchanged up to the lesser of the new and old sizes. If the new size of the memory object would require movement of the object, the space for the previous instantiation of the object is freed. If the new size is larger, the contents of the newly allocated portion of the object are unspecified. If uNewSize is 0 and pvBase is not a null pointer, the object pointed to is freed. If the space cannot be allocated, the object remains unchanged.

Upon successful completion with a uNewSize not equal to 0, the function returns a pointer to the (possibly moved) allocated space. If there is not enough available memory, the functions returns a null pointer

IN – pvBase – pointer to buffer to resize

IN – uNewSize – byte size of new buffer

5.1.9. PVRSRVCallocUserModeMem

```
IMG_PVOID PVRSRVCallocUserModeMem(IMG_SIZE_T uiSize)
```

This function allocates memory in the address space of the calling application, clearing the contents to 0

IN – ui32Size – byte size of buffer to allocate

5.1.10. PVRSRVCreateMutex

```
PVRSRV_ERROR PVRSRVCreateMutex(PVRSRV_MUTEX_HANDLE *phMutex)
```

This function creates unique per process mutex.

IN – phMutex- handle to the created mutex

RETURN

Success PVRSRV_OK

Failure PVRSRV_ERROR_INIT_FAILURE

5.1.11. PVRSRVDestroyMutex

```
PVRSRV_ERROR PVRSRVDestroyMutex (PVRSRV_MUTEX_HANDLE hMutex)
```

This function destroys per process mutex

IN – hMutex- mutex handle

RETURN

Success – valid pointer to user allocated buffer

Failure – NULL

5.1.12. PVRSRVLockMutex

```
PVRSRV_ERROR PVRSRVUnlockMutex (PVRSRV_MUTEX_HANDLE hMutex)
```

Unlocks the per process mutex

IN – Mutex– mutex handle

Success – PVRSRV_OK

Failure – PVRSRV_ERROR_GENERIC

5.1.13. PVRSRVUnlockMutex

```
PVRSRV_ERROR PVRSRVUnlockMutex (PVRSRV_MUTEX_HANDLE hMutex)
```

Unlocks the per process mutex

IN – Mutex– mutex handle

Success – PVRSRV_OK

Failure – PVRSRV_ERROR_GENERIC

5.1.14. PVRSRVEventObjectWait

```
PVRSRV_ERROR PVRSRVEventObjectWait (const PVRSRV_CONNECTION *psConnection,
                                    IMG_HANDLE hOSEvent)
```

Wait for given event object.

IN – psConnection– pointer to services connection

IN – hOSEvent– Event handle

Success – PVRSRV_OK

Failure – PVRSRV_ERROR_GENERIC

5.1.15. OSEventObjectOpen

```
                                    PVRSRV_EVENTOBJECT *psEventObject,
                                    IMG_HANDLE *phOSEvent)
```

Opens the event object.

IN – psConnection– pointer to services connection

IN – psEventObject – pointer to the event object data

OUT – phOSEvent– pointer to event object handle

Success – PVRSRV_OK

Failure – PVRSRV_ERROR_GENERIC

5.1.16. OSEventObjectClose

```
PVRSRV_ERROR PVRSRVEventObjectClose(PVRSRV_EVENTOBJECT *psEventObject,
                                    IMG_HANDLE hOSEvent)
```

Close the event object.

IN – psConnection– pointer to services connection

IN – psEventObject – pointer to the event object data

OUT – hOSEvent– handle to event object

Success – PVRSRV_OK

Failure – PVRSRV_ERROR_GENERIC

5.1.17. OSIsProcessPrivileged

```
IMG_BOOL OSIsProcessPrivileged(IMG_VOID)
```

Determines whether the calling process is ‘privileged’

RETURN

IMG_TRUE – process is privileged

IMG_FALSE – process is not privileged

This API is used with priority based hardware scheduling. The implementation of this function must be able to differentiate between privileged and non-privileged processes, where privileged processes can select raised priority levels for their associated hardware operations.

Note: As a precaution the OS implementations provided in the DDK have the default behaviour of always returning IMG_FALSE.

5.1.18. OSFlushCPUCacheRange

```
PVRSRV ERROR OSFlushCPUCacheRange(IMG VOID *pvRangeAddrStart,
                                  IMG_VOID *pvRangeAddrEnd)
```

Performs a range based CPU data cache flush. Is an optional (Usermode) fast path call for operating systems that support it (kernel function equivalent will be used if this is not supported)

IN – pvRangeAddrStart – CPU virtual range starting address

IN – pvRangeAddrEnd – CPU virtual range ending address

RETURN

Success – PVRSRV_OK

Not supported by OS - PVRSRV_ERROR_NOT_SUPPORTED

Failure – PVRSRV_ error code

5.1.19. PVRSRVSetLocale

```
IMG_CHAR * PVRSRVSetLocale(const IMG_CHAR *pszLocale)
```

Sets the 'locale' (see posix setlocale). If NULL, returns current locale.

IN – pszLocale– locale

RETURN

– Locale if pszLocale == NULL

5.2. Kernel Services

5.2.1. OSAllocMem

```
PVRSRV_ERROR OSAllocMem(
               IMG_UINT32 ui32Flags,
               IMG UINT32 ui32Size,
               IMG PVOID *ppvLinAddr,
               IMG_HANDLE *phBlockAlloc,
                IMG_CHAR *pszLogStr)
```

This function allocates OS system memory for driver data structures and buffers. The allocated memory is not intended to be addressed by 'services managed devices', should be 'cached' where possible and is only accessible in the kernel.

IN - ui32Flags

PVRSRV_OS_PAGEABLE_HEAP

Memory is pageable

PVRSRV_OS_NON_PAGEABLE_HEAP

Memory is non-pageable. Allocations of this type are required if they are to be accessed by L/MISRs

IN - ui32Size - Number of bytes to allocate

OUT - ppvLinAddr - User Mode Pointer to memory

OUT - phBlockAlloc will point to storage for an implementation specific handle which will be passed to the free routine when this block is freed

IN – pszLogStr – call logging string

RETURN

– success - PVRSRV_OK

– failure – PVRSRV_ error code

5.2.2. OSFreeMem

```
PVRSRV ERROR OSFreeMem(
        IMG_UINT32 ui32Flags,
        IMG_UINT32 ui32Size,
        IMG_PVOID pvLinAddr,
        IMG_HANDLE hBlockAlloc)
```

Frees an allocation which was created by a call to OSAllocMem

5.2.3. OSAllocPages

```
PVRSRV ERROR OSAllocPages(
        IMG UINT32 ui32Flags,
        IMG UINT32 ui32Size,
        IMG_UINT32 ui32PhysicalPageSize
        IMG_PVOID *ppvLinAddr,
        IMG_HANDLE *phBlockAlloc)
```

This function allocates OS memory which can be accessed from the hardware from a given heap

IN - ui32Flags

PVRSRV_HAP_SINGLE_PROCESS

The CPU virtual mapping of the memory is in the usermode address space of the calling process

PVRSRV_HAP_KERNEL_ONLY

The CPU virtual mapping of the memory is in the kernel address space

PVRSRV_HAP_MULTI_PROCESS

The CPU virtual mapping of the memory is in the kernel address space but can be subsequently mapped to arbitrary processes

also

PVRSRV_HAP_CACHED

Memory to be allocated as cached

PVRSRV_HAP_UNCACHED

Memory to be allocated as uncached

PVRSRV_HAP_WRITECOMBINE

Memory to be allocated as write-combined (or equivalent buffered/burst writes mechanism)

IN - ui32Size - Number of bytes to allocate

IN - ui32PhysicalPageSize – physical page size in bytes (typically 4k)

OUT – ppvLinAddr - pointer to memory

OUT – phBlockAlloc - Points to storage for an implementation specific handle

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

5.2.4. OSFreePages

```
PVRSRV ERROR OSFreePages(
        IMG UINT32 ui32Flags,
        IMG_UINT32 ui32Size,
        IMG_PVOID pvLinAddr,
        IMG_HANDLE hBlockAlloc)
```

Frees an allocation which was created by a call to OSAllocPages

5.2.5. OSInstallDeviceLISR

```
PVRSRV ERROR OSInstallDeviceLISR (
        IMG_VOID *pvSysData,
        IMG_UINT32 ui32Irq,
        const IMG_CHAR *pszISRName,
        IMG_VOID *pvDeviceNode)
```

Install a 'Device LISR'

IN - pvSysData – void pointer to Services SysData structure

IN - ui32Irq – The interrupt IRQ to be associated with the ISR routine

IN – pszISRName - name for the routine

IN – pvDeviceNode – void pointer to Services Device Node structure

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

5.2.6. OSUninstallDeviceLISR

```
PVRSRV_ERROR OSUninstallDeviceLISR(IMG_VOID *pvSysData)
```

Uninstall a 'Device LISR'

IN - pvSysData – void pointer to Services SysData structure

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

5.2.7. OSInstallSystemLISR

```
PVRSRV_ERROR OSInstallSystemLISR (
        IMG_VOID *pvSysData,
        IMG_UINT32 ui32Irq)
```

Install a 'System LISR'

IN - pvSysData – void pointer to Services SysData structure

IN - ui32Irq – The interrupt IRQ to be associated with the ISR routine

IN – pszISRName - name for the routine

IN – pvDeviceNode – void pointer to Services Device Node structure

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

5.2.8. OSUninstallSystemLISR

```
PVRSRV_ERROR OSUninstallSystemLISR(IMG_VOID *pvSysData)
```

Uninstall a 'System LISR'

IN - pvSysData – void pointer to Services SysData structure

IN – pvDeviceNode – void pointer to Services Device Node structure

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

5.2.9. OSInstallMISR

```
PVRSRV_ERROR OSInstallMISR (IMG_VOID *pvSysData)
```

Install a MISR

IN - pvSysData – void pointer to Services SysData structure

RETURN

– success - PVRSRV_OK
– failure – PVRSRV_ error code

5.2.10. OSUninstallMISR

```
PVRSRV_ERROR OSUninstallMISR(IMG_VOID *pvSysData)
```

Uninstall a MISR

IN - pvSysData – void pointer to Services SysData structure

RETURN

– success - PVRSRV_OK
– failure – PVRSRV_ error code

5.2.11. OSScheduleMISR

```
PVRSRV_ERROR OSScheduleMISR(IMG_VOID *pvSysData)
```

Schedule a MISR

IN - pvSysData – void pointer to Services SysData structure

RETURN

– success - PVRSRV_OK
– failure – PVRSRV_ error code

5.2.12. OSMemCopy

```
IMG VOID OSMemCopy(
        IMG VOID *pvDst,
        IMG_VOID *pvSrc,
        IMG_SIZE_T uiSize)
```

This function is an abstraction of the ANSI memcpy() function.

Copy a (non-overlapping) block of memory of ui32Size bytes from pvSrc to pvDst

IN – pvDst - Destination block of memory

IN – pvSrc - Source block of memory

IN - ui32Size - Number of bytes to copy

5.2.13. OSMemSet

```
IMG VOID OSMemSet(
        IMG_VOID *pvDest,
        IMG_UINT8 ui8Value,
        IMG_SIZE_T uiSize)
```

This function is an abstraction of the ANSI memset() function.

Set ui32Size bytes from pvSrc to the value ui8Value

IN – pvDest - Pointer to destination (which will be overwritten)

IN - ui8Value - Value to write to destination

IN - uiSize - Number of bytes to set

5.2.14. OSStringCopy

```
IMG_CHAR* OSStringCopy(
        IMG_CHAR *pszDest,
        Const IMG_CHAR *pszSrc)
```

This function is an abstraction of the ANSI strcpy() function.

Copies a zero terminated string from pszSrc into pszDest

IN – pszDest - Memory for the string to be copied into

IN – pszSrc - Zero terminated source string

RETURN - Returns pointer to pszDest

5.2.15. OSCreateResource

```
PVRSRV_ERROR OSCreateResource (
PVRSRV_RESOURCE *psResource)
```

Creates an OS dependant resource object. The resource object allows services code to implement exclusive execution of critical code as well as atomic acquisition of arbitrary resources in an OS agnostic manner.

IN - psResource - pointer to an implementation dependent resource

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

5.2.16. OSDestroyResource

```
PVRSRV_ERROR OSDestroyResource (
        PVRSRV_RESOURCE *psResource)
```

Destroys an OS dependant resource object

IN - psResource - pointer to the data returned from a call to OSCreateResource()

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

5.2.17. OSLockResource

```
PVRSRV ERROR OSLockResource (
        PVRSRV_RESOURCE *psResource,
        IMG_UINT32 ui32ID)
```

This function locks an OS dependant resource. The implementation of the lock can be considered to be atomic or, in practical terms, guaranteed to function correctly in the case of two or more threads of execution contending for the same resource

IN – psResource - pointer to the data returned from a call to OSCreateResource()

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

5.2.18. OSUnlockResource

```
PVRSRV ERROR OSUnlockResource (
        PVRSRV RESOURCE *psResource,
        IMG_UINT32 ui32ID)
```

Unlocks an OS dependant resource

IN – psResource - pointer to the data returned from a call to OSCreateResource()

IN – ui32ID – lock ID, generally a process/threadID

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

5.2.19. OSBreakResourceLock

```
IMG_VOID OSBreakResourceLock (
        PVRSRV RESOURCE *psResource,
        IMG_UINT32 ui32ID)
```

Forces the unlock of an OS dependant resource

IN – psResource - pointer to the data returned from a call to OSCreateResource()

IN – ui32ID – lock ID, generally a process/threaded

5.2.20. OSIsResourceLocked

```
IMG BOOL OSIsResourceLocked (
        PVRSRV_RESOURCE *psResource,
        IMG_UINT32 ui32ID)
```

Tests if resource is locked

IN – psResource - pointer to the data returned from a call to OSCreateResource()

IN – ui32ID – lock ID, generally a process/threadID

RETURN - IMG_TRUE if locked by the given ID, otherwise IMG_FALSE

5.2.21. OSEventObjectCreate

```
PVRSRV_ERROR OSEventObjectCreate (
        const IMG_CHAR *pszName,
        PVRSRV_EVENTOBJECT *psEventObject)
```

Destroys an event object

IN – pszName - Globally unique event object name (if null name must be autogenerated)

IN – psEventObject - OS event object info structure

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

5.2.22. OSEventObjectDestroy

```
PVRSRV_ERROR OSEventObjectDestroy (
        IMG_HANDLE hOSEventKM))
```

Forces the unlock of an OS dependant resource

IN – psEventObject - OS event object info structure

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

5.2.23. OSEventObjectSignal

```
IMG BOOL OSEventObjectSignal (
        PVRSRV_RESOURCE *psResource,
        IMG_UINT32 ui32ID)
```

Signal an event object. Called from L/MISR

IN – hOSEventKM - OS and kernel specific handle to event object

IN – ui32ID – lock ID, generally a process/threadID

- failure – PVRSRV_ error code

5.2.24. OSEventObjectWait

```
IMG_BOOL OSEventObjectWait (
        IMG_HANDLE hOSEventKM)
```

Wait on an event object. This function is required on some OS implementations to support equivalent calls from the client.

IN – hOSEventKM - OS and kernel specific handle to event object

- failure – PVRSRV_ error code

5.2.25. OSEventObjectOpen

```
IMG_BOOL OSEventObjectOpen (
        PVRSRV_EVENTOBJECT psEventObject,
        IMG_HANDLE *phOSEventKM)
```

Open an event object. . This function is required on some OS implementations to support equivalent calls from the client.

IN – psEventObject – event object information

OUT – phOSEventKM – pointer to OS and kernel specific handle to event object

– failure – PVRSRV_ error code

5.2.26. OSEventObjectClose

```
IMG BOOL OSEventObjectOpen (
        PVRSRV_EVENTOBJECT psEventObject,
        IMG_HANDLE hOSEventKM)
```

Close an event object. This function is required on some OS implementations to support equivalent calls from the client.

IN – psEventObject – event object information

OUT – hOSEventKM – OS and kernel specific handle to event object

failure – PVRSRV_ error code

5.2.27. OSMapLinToCPUPhys

```
IMG_CPU_PHYADDR OSMapLinToCPUPhys(IMG_VOID* pvLinAddr)
```

This function returns a CPU relative physical bus address given a linear CPU address.

Note: address passed in is linear rather virtual, i.e. the underlying physical pages that back the linear address range are contiguous.

IN – pvLinAddr - pointer to locked location

RETURN - CPU relative physical bus address corresponding to the passed linear address

5.2.28. OSGetCurrentProcessID

```
IMG_UINT32 OSGetCurrentProcessID (IMG_VOID)
```

This function returns a unique identifier for the current process

RETURN - ID of current process

5.2.29. OSMapPhysToLin

```
IMG_PVOID OSMapPhysToLin (
        IMG_CPU_PHYADDR BasePAddr,
        IMG_SIZE_T uiBytes,
        IMG_UINT32 ui32CacheType,
        IMG_HANDLE *phPageAlloc)
```

This function maps a range physical memory to a CPU linear address range

Note: the address returned is linear rather virtual, i.e. the underlying physical pages corresponding to the CPU physical address are contiguous.

IN - BasePAddr - physical cpu address to map

IN - uiBytes - number of bytes to map

IN - ui32CacheType:

CACHETYPE_WRITECOMBINED

CACHETYPE_CACHED

CACHETYPE_UNCACHED

IN - phPageAlloc - pointer to a mapping handle

RETURN – CPU virtual address of mapping on success, else NULL

5.2.30. OSUnMapPhysToLin

```
IMG BOOL OSUnMapPhysToLin(
        IMG VOID *pvLinAddr,
        IMG_SIZE_T uiBytes,
        IMG_HANDLE hPageAlloc)
```

This function unmaps memory that was mapped with OSMapPhysToLin

IN – pvLinAddr - pointer returned by OSMapPhysToLin

IN – uiBytes - number of bytes to unmap

IN - hPageAlloc - mapping handle

RETURN - IMG_TRUE on success, IMG_FALSE on failure

5.2.31. OSClockus

```
IMG_UINT32 OSClockus (IMG_VOID)
```

Returns clock value in microseconds

RETURN – clock time in microseconds

5.2.32. OSWaitus

```
IMG_VOID OSWaitus (
        IMG_UINT32 ui32Timeus)
```

This function waits ui32Timeus microseconds (does NOT release thread quanta).

Note: The accuracy of this function is not essential as it is used in conjunction with OSClockus which should provide accurate timing (at least in terms of milliseconds).

IN – ui32Timeus – time to wait in microseconds

5.2.33. OSInitEnvData

```
PVRSRV ERROR OSInitEnvData (
        IMG_PVOID *ppvEnvSpecificData)
```

Allocates space for environment specific data

IN – ppvEnvSpecificData – double pointer to OS specific data

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

5.2.34. OSDeInitEnvData

```
PVRSRV_ERROR OSDeInitEnvData (
        IMG_PVOID pvEnvSpecificData)
```

Frees environment specific data memory

IN – ppvEnvSpecificData – pointer to OS specific data

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

5.2.35. OSAddTimer

```
IMG_HANDLE OSAddTimer(
PFN_TIMER_FUNC pfnTimerFunc,
        IMG_VOID *pvData,
        IMG_UINT32 ui32MsTimeout)
```

Implementation specific function to install a timer callback function

IN – pfnTimerFunc – function to be called on timeout

IN – pvData – pointer to data to be passed to callback function

IN - ui32MsTimeout - callback timeout period in milli seconds

RETURN - valid handle success, NULL failure

5.2.36. OSRemoveTimer

```
PVRSRV ERROR OSRemoveTimer(
        IMG_HANDLE hTimer)
```

Remove timer installed with OSAddTimer() function.

IN – hTimer – handle to timer

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

5.2.37. OSEnableTimer

```
PVRSRV_ERROR OSEnableTimer(
        IMG_HANDLE hTimer)
```

Enable timer installed with OSEnableTimer() function.

IN – hTimer – handle to timer

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

5.2.38. OSDisableTimer

```
PVRSRV_ERROR OSDisableTimer(
        IMG_HANDLE hTimer)
```

Disable timer installed with OSDisableTimer() function.

IN – hTimer – handle to timer

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

5.2.39. OSWriteHWReg

```
IMG VOID OSWriteHWReg(IMG PVOID pvLinRegBaseAddr,
        IMG_UINT32 ui32Offset,
        IMG_UINT32 ui32Value)
```

Register write function

IN – pvLinRegBaseAddr – CPU linear address of the base of the register block to write to

IN – ui32Offset – bytes offset into the register block

IN – ui32Value – 32bit values to write

RETURN - none

5.2.40. OSReadHWReg

```
IMG UINT32 OSWriteHWReg(IMG PVOID pvLinRegBaseAddr,
        IMG_UINT32 ui32Offset)
```

Register read function

IN – pvLinRegBaseAddr – CPU linear address of the base of the register block to read from

IN – ui32Offset – bytes offset into the register block

IN – ui32Value – 32bit values to write

RETURN

value read back from the register

5.2.41. OSProcHasPrivSrvInit

```
IMG_BOOL OSProcHasPrivSrvInit(IMG_VOID)
```

Checks if the process has sufficient privileges to initialise services.

RETURN

- TRUE - has privileges
- FALSE - has not

5.2.42. OSCopyToUser

```
PVRSRV_ERROR OSCopyToUser(IMG_VOID *pvDest,
        IMG_VOID *pvSrc,
        IMG_SIZE_T uiBytes)
```

Copy a block of data into user space

IN – pvDst - Destination block of memory

IN – pvSrc - Source block of memory

IN - uiBytes- Number of bytes to copy

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

5.2.43. OSCopyFromUser

```
PVRSRV_ERROR OSCopyFromUser(IMG_VOID *pvDest,
        IMG_VOID *pvSrc,
        IMG_SIZE_T uiBytes)
```

Copy a block of data from the user space

IN – pvDst - Destination block of memory

IN – pvSrc - Source block of memory

IN - uiBytes- Number of bytes to copy

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

5.2.44. OSAccessOK

```
PVRSRV_ERROR OSAccessOK(IMG_VERIFY_TEST eVerification,
        IMG_VOID *pvUserPtr,
        IMG_SIZE_T uiBytes)
```

Checks if a user space pointer is valide

IN – eVerification- Read of write permission to the pointer

IN – pvUserPtr- User pointer

IN - uiBytes- Number of bytes to copy

RETURN

- IMG_BOOL

5.2.45. OSPanic

```
IMG_VOID OSPanic(IMG_VOID)
```

Performs an OS specific ‘panic’ in response to some critical failure and allowing for post-mortem debug

RETURN

- none

5.2.46. OSFlushCPUCacheKM

```
IMG_VOID OSFlushCPUCacheKM()
```

Performs a full CPU data cache flush

RETURN

– None

5.2.47. OSCleanCPUCacheKM

```
IMG_VOID OSFlushCPUCacheKM()
```

Performs a full CPU data cache Clean

RETURN

– None

5.2.48. OSFlushCPUCacheRangeKM

```
IMG BOOL OSFlushCPUCacheRangeKM(IMG HANDLE hOSMemHandle,
                                IMG_VOID *pvRangeAddrStart,
                                IMG_UINT32 ui32Length)
```

Performs a range based CPU data cache flush. an optional (Usermode) fast path call is provided for operating systems that support it (kernel function equivalent will be used if this is not supported)

IN – hOSMemHandle – Implementation specific handle obtained by calling OSAllocMem

IN – pvRangeAddrStart – CPU virtual starting address

IN – ui32Length – Size of the region to flush

RETURN

– IMG_BOOL

5.2.49. OSCleanCPUCacheRangeKM

```
IMG VOID OSCleanCPUCacheRangeKM(IMG HANDLE hOSMemHandle,
                                IMG_VOID *pvRangeAddrStart,
                                IMG_UINT32 ui32Length)
```

Performs a range based CPU data cache clean. Provides an optional (Usermode) fast path call for operating systems that support it (kernel function equivalent will be used if this is not supported)

IN – hOSMemHandle – Implementation specific handle obtained by calling OSAllocMem

IN – pvRangeAddrStart – CPU virtual starting address

IN – ui32Length – Size of the region to flush

RETURN - None

5.2.50. OSInvalidateCPUCacheRangeKM

```
IMG_VOID OSInvalidateCPUCacheRangeKM(IMG_VOID *pvRangeAddrStart,
                                     IMG_VOID *pvRangeAddrEnd)
```

Performs a range based CPU data cache Invalidate. Provides an optional (Usermode) fast path call for operating systems that support it (kernel function equivalent will be used if this is not supported)

IN – hOSMemHandle – Implementation specific handle obtained by calling OSAllocMem

IN – pvRangeAddrStart – CPU virtual starting address

IN – ui32Length – Size of the region to flush

RETURN - None

6. System Functions

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*.

Note: If an SOC device does not use a specific sys function, it must implement a stub.

To find the most suitable implementation of a given *sys* function (on which to base the new SOC's equivalent function), it can be helpful to inspect several *sysconfig.c* files.

Note: Some systems may include additional functions for use within the system component only.

6.1.1. 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

6.1.2. 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

6.1.3. SysFinalise

```
PVRSRV_ERROR SysFinalise ()
```

This function completes the final stage of initialisation. Typically this function installs the MISR and LISR.

RETURN

- success - PVRSRV_OK
- failure – PVRSRV_ error code

6.1.4. 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

6.1.5. 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

6.1.6. SysCpuPAddrToDevPAddr

```
IMG_DEV_PHYADDR SysCpuPAddrToDevPAddr (IMG_CPU_PHYADDR CpuPAddr)
```

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

IN – CpuPAddr – CPU physical address

RETURN – device physical address

6.1.7. 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

6.1.8. 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

6.1.9. SysSysPAddrToDevPAddr

```
IMG_DEV_PHYADDR SysSysPAddrToDevPAddr (IMG_SYS_PHYADDR SysPAddr)
```

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

IN – SysPAddr – CPU physical address

RETURN – device physical address

6.1.10. SysDevPAddrToSysPAddr

```
IMG_SYS_PHYADDR SysDevPAddrToSysPAddr (IMG_DEV_PHYADDR DevPAddr)
```

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

IN – DevPAddr – Device physical address

RETURN – device physical address

6.1.11. SysRegisterExternalDevice

```
IMG_VOID SysRegisterExternalDevice (PVRSRV_DEVICE_NODE *psDeviceNode)
```

This function is called when an external third party device registers with Services. This function has no implementation unless the system configuration uses a System ISR (see OSInstallSystemLISR). In this case, the function must specify which bit in psDeviceNode->ui32SOCInterruptBit corresponds to this device

IN – psDeviceNode – device node of the 3rd party device

RETURN – none

6.1.12. SysRemoveExternalDevice

```
IMG_VOID SysRemoveExternalDevice (PVRSRV_DEVICE_NODE *psDeviceNode)
```

This function is called when an external third party device unregisters from Services. Generally this function has no implementation

IN – psDeviceNode – device node of the 3rd party device

RETURN – none

6.1.13. 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.

If custom OEM functions are not required, implement a stub function.

If custom OEM functions are required, implement a switch statement to process the ui32ID parameter.

6.1.14. SysSystemPrePowerState

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

IN – eNewPowerState – New system power state, see PVRSRV_POWER_STATE_*

SysSystemPrePowerState provides notification of a change in **system** power state which is about to happen. On systems where power to devices is controlled by underlying drivers, e.g. PCI, this function will be called **before** the physical power change event.

When a system power state change event happens, device state change events will also be generated if required, i.e. SysDevicePrePowerState will be called, for each device, just before SysSystemPrePowerState. Therefore, SysSystemPrePowerState should perform only actions which are specific to system power events, e.g. backing up the local memory on a PCI system.

If necessary, the old system power state can be queried using the eCurrentPowerState field of the Services SysData structure. However, this field should not be written by system functions.

As with all *sys* functions, implement a stub function if unused.

6.1.15. SysSystemPostPowerState

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

IN – eNewPowerState – New system power state, see PVRSRV_POWER_STATE_*

SysSystemPostPowerState provides notification of a change in **system** power state which has just happened. On systems where power to devices is controlled by underlying drivers, e.g. PCI, this function will be called **after** the physical power change event.

When a system power state change event happens, device state change events will also be generated if required, i.e. SysDevicePostPowerState will be called, for each device, just after SysSystemPostPowerState. Therefore, SysSystemPostPowerState should perform only actions which are specific to system power events, e.g. restoring the local memory on a PCI system.

If necessary, the old system power state can be queried using the eCurrentPowerState field of the Services SysData structure. However, this field should not be written by system functions.

As with all *sys* functions, implement a stub function if unused.

6.1.16. SysDevicePrePowerState

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

IN – ui32DeviceIndex – Index of the device whose power state is changing

IN – eNewPowerState – New device power state, see PVRSRV_POWER_STATE_*

IN – eCurrentPowerState – Old device power state, see PVRSRV_POWER_STATE_*

SysDevicePrePowerState provides pre-notification of a change in **device** power state. SysDevicePrePowerState will be called after the device has been paused (for power down events). If the device power state change event is happening as a result of a system power state change, SysDevicePrePowerState will be called before SysSystemPrePowerState.

If applicable to the system, SysDevicePrePowerState is the place to change an SOC device's power mode when the device must be powered down – generally by performing appropriate register-writes.

6.1.17. SysDevicePostPowerState

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

IN – ui32DeviceIndex – Index of the device whose power state is changing

IN – eNewPowerState – New device power state, see PVRSRV_POWER_STATE_*

IN – eCurrentPowerState – Old device power state, see PVRSRV_POWER_STATE_*

SysDevicePostPowerState provides post-notification of a change in **device** power state. SysDevicePostPowerState will be called before the device has been re-initialised (for power up events). If the device power state change event is happening as a result of a system power state change, SysDevicePrePowerState will be called after SysSystemPostPowerState.

If applicable to the system, SysDevicePrePowerState is the place to change an SOC device's power mode when the device must be powered up – generally by performing appropriate register-writes.

6.1.18. SysGetInterruptSource

```
IMG_UINT32 SysGetInterruptSource(SYS_DATA* psSysData, PVRSRV_DEVICE_NODE *psDeviceNode)
```

SysGetInterruptSource returns system specific information about the device(s) that generated the interrupt.

6.1.19. SysClearInterrupts

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

SysClearInterrupts clears system(SOC) specific interrupts specified by ui32ClearBits

7. Constants & Definitions

7.1. Sysconfig.h

Sysconfig.h provides system-specific declarations and macros.

Note: Declarations specific to 3rdparty display devices should not be included in sysconfig.h – because other users of this file may be using the same system with different displays.

This is a summary of the constants likely to be required in sysconfig.h. This list is not intended to be exhaustive and will vary from system to system.

7.1.1. Register size

SGX_REG_SIZE

The size of the SGX global register block in bytes. This is used is specifying the physical address map of the device

SGX_SP_SIZE

The size of the SGX 2D slave port write range in bytes. This is used is specifying the physical address map of the device

7.1.2. Platform Name String

VS_PRODUCT_NAME

The SOC product name. This is a string, e.g. "SGX No HW"

7.2. Sysinfo.h

Sysinfo.h contains a list of definitions and declarations. It is not intended to be an exhaustive list.

7.2.1. System specific poll and timeout details

MAX_HW_TIME_US

This constant represents the duration of the longest possible single task that can submitted on a device managed by services in a given system. It is used to trigger time-out code at various points in the drivers.

WAIT_TRY_COUNT

This constant represents how many times polling code should test for completion status during the period MAX_HW_TIME_US, i.e. every MAX_HW_TIME_US/ WAIT_TRY_COUNT microseconds.

7.2.2. Device types

```
typedef enum  SYS DEVICE TYPE
{
        SYS DEVICE SGX = 0,
        SYS_DEVICE_FORCE_I16 = 0x7fff
} SYS_DEVICE_TYPE;
```

This enumeration specifies device types that may exist in a given system

SYS_DEVICE_COUNT

This is a count of devices present in a given system. Note that this count should include any 3rd party display class or buffer class devices that may register with services at boot time.

7.2.3. Command types

Services code provides interrupt driven command processing functionality in software to allow devices to queue commands for asynchronous execution. Some devices require this functionality and some do not. Generally, SGX manages command processing internally within its programmable hardware. However, SGX integration with 3rd party display controllers (flip commands) requires the command processor functionality to synchronise the renders and display flips.

```
typedef enum _SGX_COMMAND_TYPE_
{
        SGX BLT COMMAND                                 = 0,
        ..
        SGX COMMAND COUNT,
        PVRSRV_ERROR_FORCE_I16                          = 0x7fff
} SGX_COMMAND_TYPE;
```

These enumerations are used to index the devices' function pointer tables for private command processing. Note that SYS_DATA has an array of function pointer tables – one table per device in the system, i.e.

```
PFN_CMD_PROC *ppfnCmdProcList[SYS_DEVICE_COUNT];
```

Each table is allocated when the device registers its private command processing functions.

7.2.4. SGX Slave Port FIFO (Subject to change)

SGX_SP_FIFO_DWSIZE

This value relates to the depth of the SGX 2D FIFO in 32bit slots.

Note: the value is also dependent on the slave port bus width.

7.3. SGX Device Map

The SGX device map is a structure containing system constants pertaining to SGX used by the services. It is passed to the services via the SysGetDeviceMemoryMap call.

Members of `SGX_TIMING_INFORMATION`:

ui32CoreClockSpeed

The speed of the SGX clock, specified in Hz.

ui32HWRecoveryFreq

The frequency at which polling for hardware lockups is performed, specified in Hz. A higher frequency means a higher responsiveness of hardware recovery, whereas a lower frequency has less performance impact.

Note that the precision of this value is dependent on the value of i32uKernelFreq – see below.

bEnableActivePM

run time control for SGX Active power management.

ui32ActivePowManLatencyms

Specified in milliseconds. The minimum time after SGX becomes idle to wait before signalling to the driver that it should be powered down. The power down signal is sent to the host using an interrupt from the periodic microkernel timer, and therefore there may be an extra delay of up to the length of the microkernel timer period.

ui32uKernelFreq

This is the frequency at which the SGX microkernel periodic timer task is run, specified in Hz. Granularity of control over the Hardware Recovery frequency and the Active Power latency is limited by this value. For example, if the microkernel timer frequency is 100Hz (i.e. the timer period is 10ms), the hardware recovery polling period will be a multiple of 10ms, and the active power signal will have a potential delay of up to 10ms.

Increasing the timer frequency may have a negative performance impact.

8. Build Configuration

This section details various porting specific build options and the options which are listed during execution of the services sgx_init_test.

8.1. SGX System Cache Feature

If SGX is configured with the System Cache feature then this should be enabled in the build by specifying the following on the build command line:

```
SUPPORT_SLC=1
```

Note: this defines the feature SGX_FEATURE_SYSTEM_CACHE

8.2. Microkernel build options

The following options can be enabled at build time (depending on hardware support) and affect the operation of the microkernel.

DEBUG

Enable debugging.

PDUMP

Enable parameter dumping, which allows IMG tools running offline to reconstruct the memory allocations and register reads and writes during a render.

PVRSRV_DUMP_MK_TRACE

Write microkernel trace to system log during HW recovery.

SUPPORT_HW_RECOVERY

Enable support for lockup detection in the microkernel (also enables lockup detection in the host driver).

PVR_SECURE_HANDLES

Enable secure handles in host driver.

SGX_BYPASS_SYSTEM_CACHE

Bypass the SLC (System Level Cache).

Note: SUPPORT_SLC and SGX_FEATURE_SYSTEM_CACHE must also be defined

SGX_DMS_AGE_ENABLE

Enable age-based task scheduling to reduce microkernel scheduling time.

SGX_FAST_DPM_INIT

Enable fast DPM initialisation on no hardware drivers. Only required for RTL testing.

SGX_FEATURE_DCU

This is enabled if the hardware supports a DCU (Data Cache Unit).

SGX_FEATURE_MP

This is enabled if the SGX has more than one core.

SGX_FEATURE_MP_CORE_COUNT

When used in conjunction with SGX_FEATURE_MP defines the number of cores available. Range is 1 to 16.

SGX_FEATURE_MULTITHREADED_UKERNEL

Internal use only.

SGX_FEATURE_SYSTEM_CACHE

This is enabled if the SGX has a SLC (System Level Cache).

SGX_SUPPORT_HWPROFILING

Internal use only.

SUPPORT_ACTIVE_POWER_MANAGEMENT

Enable power management within the microkernel, which allows the PVR HW to power down if there is no more work to process.

SUPPORT_DISPLAYCONTROLLER_TILING

Enable support for tiled memory layout on 3rd party display controller. The tiled address translation unit on SGX is also enabled for surfaces that fall into the SGX tiled addressing ranges.

SUPPORT_PERCONTEXT_PB

Enable a per-client parameter buffer. If unset then the parameter buffer is shared between all client applications.

SUPPORT_SGX_HWPERF

Enable SGX performance profiling data for analysis with IMG PVR tools such as PVRScope.

SUPPORT_SGX_MMU_DUMMY_PAGE

Enable dummy data page allocation for uninitialised PTEs to point to, for debugging purposes.

SUPPORT_SGX_PRIORITY_SCHEDULING

Enable priority-based task scheduling. The implementation details depend on HW support.

USE_SUPPORT_NO_TA3D_OVERLAP

Disable concurrent execution of the TA and 3D. Use only for debugging and testing purposes as there is an obvious performance hit.

Appendix A. Display Controllers

8.3. Display Controllers Overview

- Important for independent display hardware to be 'coordinated' with services devices
- 3rd party display API provides a consistent interface between services and 3rd Party display device drivers
- Abstracts control of display hardware via the Display Class API (used by OGLES, D3DM, DDraw, windowing system driver, etc.)
- Provides maximum performance while maintaining the order of operations on shared resources.

8.4. Display Class API architecture

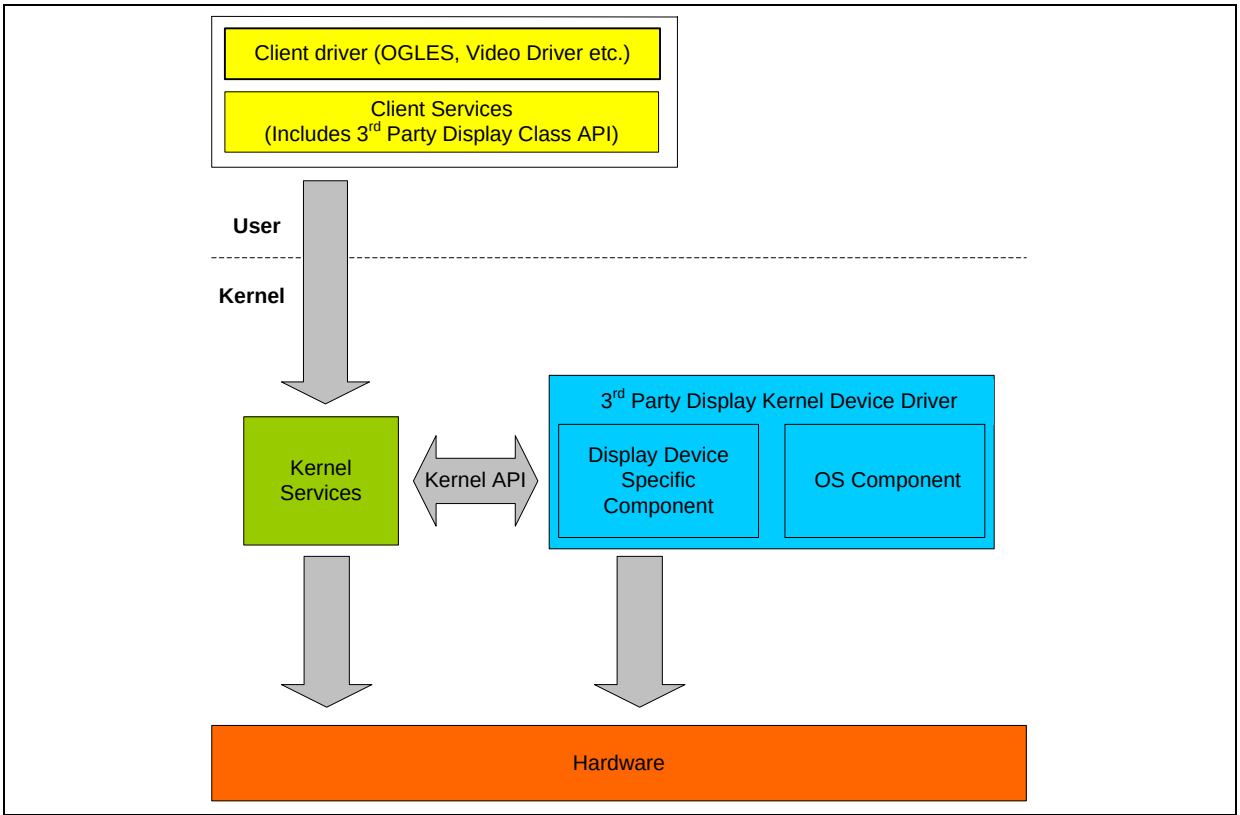


Figure 4 Display Class Architecture

8.5. Adding a new display device

Adjust the SYS_DEVICE_COUNT in /system/<SOCname>/sysinfo.h to include the new display

Create new display device folders in:

- /3rdparty
- /3rdparty/<displayname>/include
- /3rdparty/<displayname>/kernel

Use an existing reference implementation as a template

Consists of a single kernel mode driver

If a similar component already exists, extend it to support the 3rd party display device API.

e.g. already have a kernel display driver so add 3rd party display functionality as part of the same driver

Note: For fully details on developing 3rd party display drivers see the 3rd party display API document

Appendix B. SGX Device Versions and Revisions

Any given build configuration of Services for an SGX Family device must specify the SGX Core version and revision. Core versions include:

```
SGX520
SGX530
SGX535
SGX540
SGX543
SGX545
```

The OS specific build configuration must set one of the versions above as a build environment variable.

Core revision is specified by the following build environment variable:

```
SGX_CORE_REV
```

The OS specific build configuration must set the core revision to a valid value. The core revision is generally a 3 digit number, e.g. 1.1.1 where the decimal points are removed in the actual definition of the variable, say

```
SGX_CORE_REV = 111
```

There is a central errata definition file for all SGX family versions (`sgxerrata.h`). This file maps all valid combinations of SGX Core version and revision to associated hardware BRN listings.

The inclusion of `sgxerrata.h` always precedes the inclusion of the SGX feature definitions header file (`sgxfeaturedefs.h`). The feature definition file specifies a list of 'features' for every SGX family core version. In the last section of the feature definition file specific features are conditionally removed (`#undef`) based on the presence of specific hardware BRNs.

B.1. Patched RTL revisions

In the case of 'patched' RTL releases where a customer takes a modified RTL version of a reference revision then a 4th digit is introduced, indicating a modification of the equivalent reference revision. The digit 0 is reserved and 1 through 9 used for every unique patched release of a given version. The digits 1 through 9 are simply unique and add no chronological detail to the revision. The first patched release of SGX 535 1.1.1 would have the following core revision:

```
SGX_CORE_REV = 1111
```

Note: when new 4 digit patched revisions are created, an equivalent BRN listing must be created in sgxerrata.h. Below is an example code snippet:

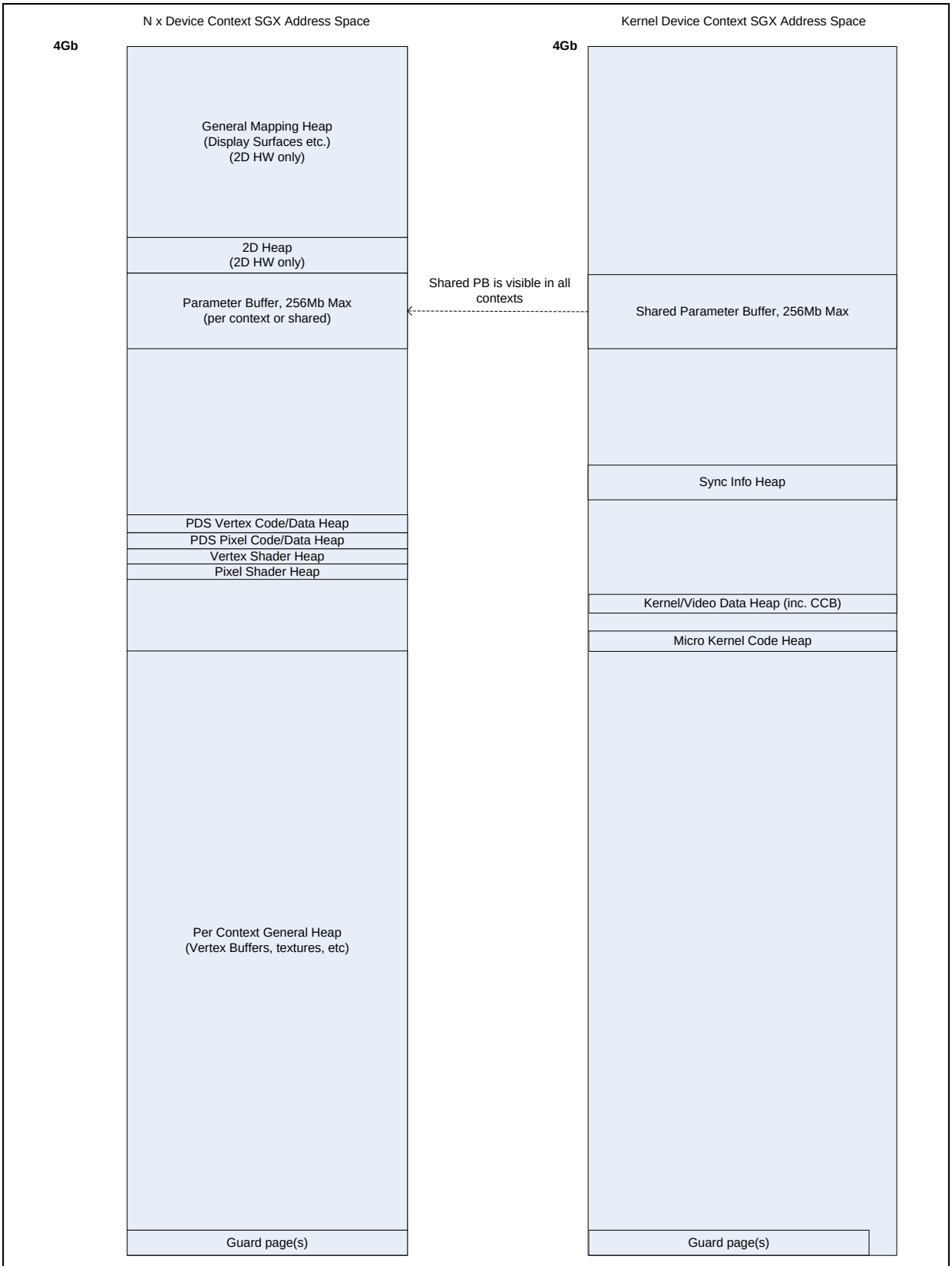
```
#if (defined(SGX535) && !defined(SGX_CORE_DEFINED)
        ...
        #if SGX CORE REV == 111
                ...
        #else
        #if SGX_CORE_REV == 1111/* 'modified' 111 */
                ...
                /* Add BRN list for 'patched' revision */
        #else
        #if SGX CORE REV == SGX CORE REV HEAD
                /* RTL head - no BRNs to apply */
        #else
                #error "sgxerrata.h: SGX535 Core Revision unspecified"
        #endif
        #endif
        #endif
#endif
```

Appendix C. Default Memory Configurations

Kernel Services specifies default memory configurations for each device it manages. Specifically, a default ‘template’ is defined specifying how many heaps should be created in order to manage a given device memory address space. Each heap description in a template is assigned a set of attributes to be used when the heaps are created.

Note: The system code (SysInitialise) is at liberty to redefine the heap configuration although generally it should not be required to do so.

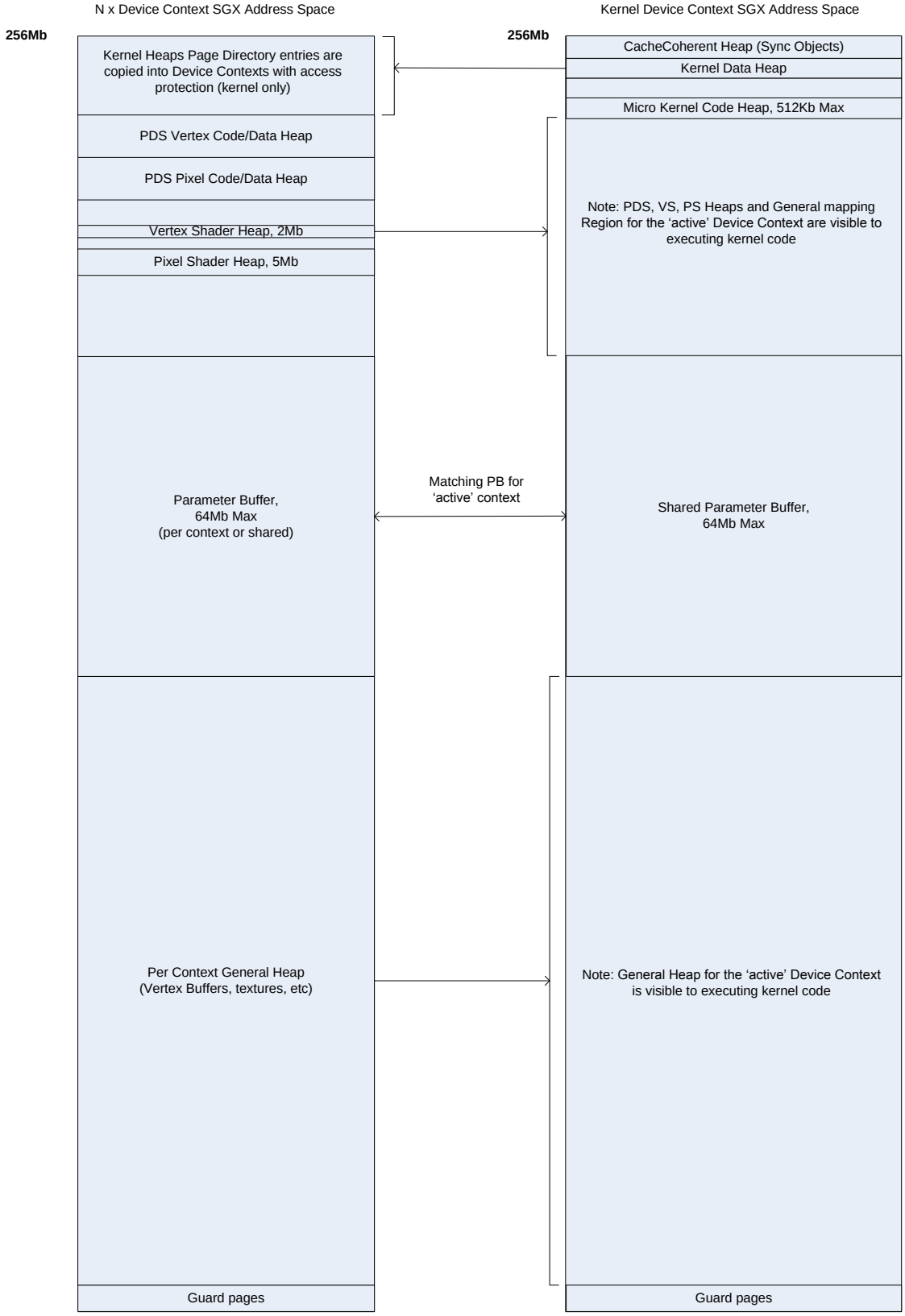
SGX535 Default Memory Configuration



Note: All heaps shall be separated by at least one invalid page for easy detection of heap overflows

Figure 5 SGX535 default memory configuration

SGX530 and SGX520 Default Memory Configuration



Note: All heaps shall be separated by at least one invalid page for easy detection of heap overflows

Figure 6 SGX530 and SGX520 default memory configuration

MSVDX Default Memory Configuration

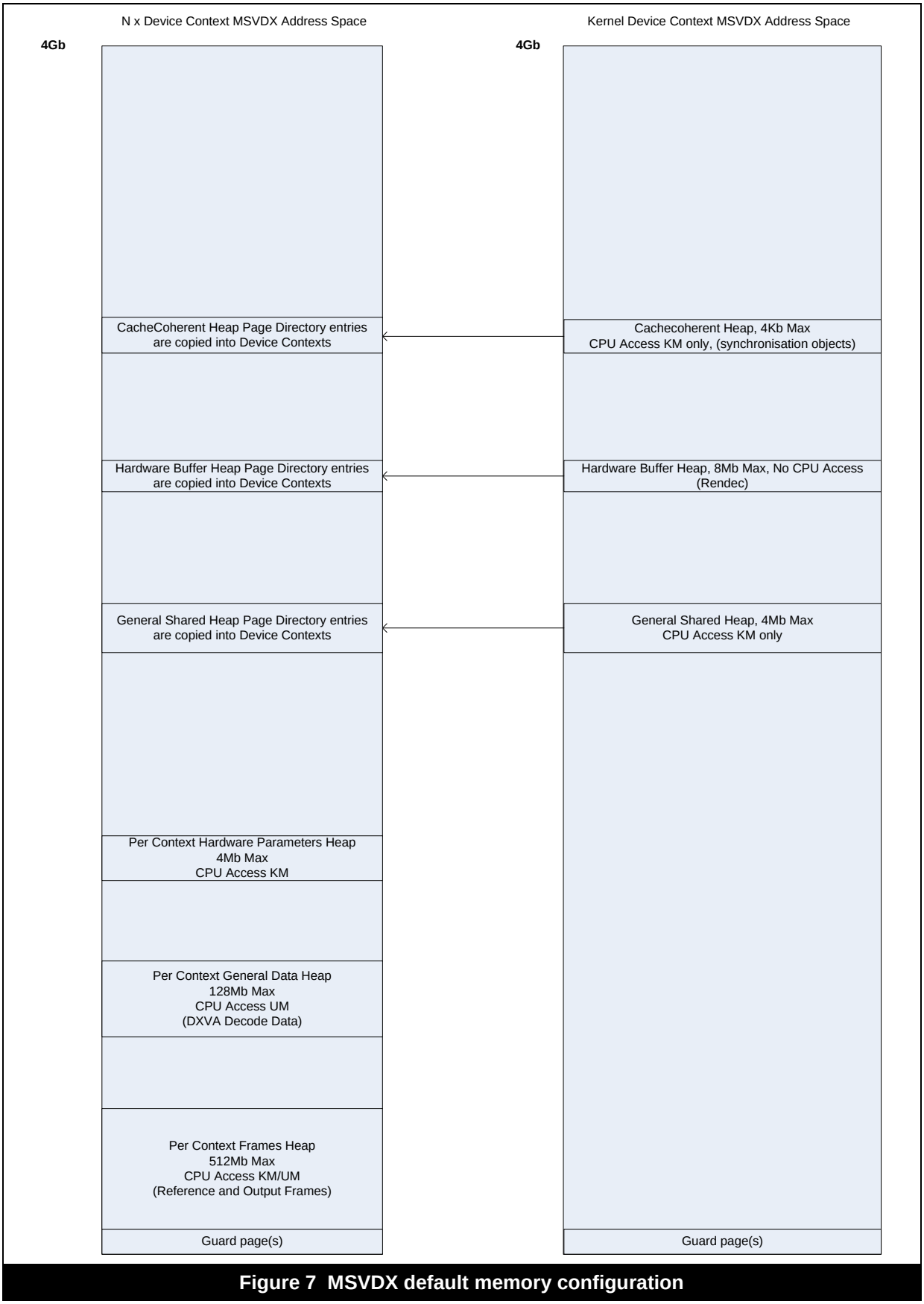


Figure 7 MSVDX default memory configuration

Appendix D. Configuring non-UMA systems

A non-UMA system has dedicated memory local to one or more devices managed by services. True UMA systems may also be configured as non-UMA by reserving a block of physical system memory outside the control of the OS or allocating a contiguous block at start-up. In either case, Services is configured in the same way for all non-UMA systems.

The services system files (sysconfig.c, sysconfig.h etc.) in services\system\<system name>\ provide the system configuration abstraction.

Physical Backing Store

Whether the non-UMA memory is true local device memory, reserved or allocated system memory, in all cases the memory provides the 'physical backing store' for SGX addressable memory.

D.1. System Configuration

The first step is to identify the system physical address of the base of the backing store memory as well as the size.

Note: in the case of PCI systems it's necessary to derive the physical base address dynamically and is based on the PCI BAR setting.

The backing store information is stored in the local static structure, gsSGXDeviceMap, and is set-up in SysLocateDevices() function in sysconfig.c:

```
        /* system physical base of backing store */
        gsSGXDeviceMap.sLocalMemSysPBase.uiAddr;
        /* size of backing store in bytes */
        gsSGXDeviceMap.ui32LocalMemSize;
```

After all devices have been registered with services device manager (i.e. after PVRSRVRegisterDevice() is called for all devices from SysInitialise() in sysconfig.c) then the 'backing store' can be integrated into services. All backing stores are managed in Services by Resource Allocators (RA for short). Any system may support up to SYS_MAX_LOCAL_DEVMEM_ARENAS independent backing stores, each of which are created with a call to the following function:

```
gpsSysData->apsLocalDevMemArena[0] = RA_Create ("SGXLocalDeviceMemory",
                                                gsSGXDeviceMap.sLocalMemSysPBase.uiAddr,
                                                gsSGXDeviceMap.ui32LocalMemSize,
                                                IMG NULL,
                                                HOST PAGESIZE(),
                                                IMG NULL,
                                                IMG_NULL,
                                                IMG_NULL,
                                                IMG_NULL);
```

Note: generally only one non-UMA backing store exists

Once all local backing store RAs have been created then the every device's heaps must be configured to use the appropriate backing store. Below is a code fragment example:

```
psDeviceNode = gpsSysData->psDeviceNodeList;
while(psDeviceNode)
{
	/* perform any OEM SOC address space customisations here */
	switch(psDeviceNode->sDevId.eDeviceType)
	{
		case PVRSRV_DEVICE_TYPE_SGX:
		{
			DEVICE_MEMORY_INFO *psDevMemoryInfo;
			DEVICE_MEMORY_HEAP_INFO *psDeviceMemoryHeap;

			/* specify the backing store to use for the devices MMU PT/PDs */
			psDeviceNode->psLocalDevMemArena = gpsSysData->apsLocalDevMemArena[0];

			/* useful pointers */
			psDevMemoryInfo = &psDeviceNode->sDevMemoryInfo;
			psDeviceMemoryHeap = psDevMemoryInfo->psDeviceMemoryHeap;

			/* specify the backing store for all SGX heaps */
			for(i=0; i<psDevMemoryInfo->ui32HeapCount; i++)
			{
				/*
					specify the backing store type
					(all local device mem noncontig) for emulator
				*/
				psDeviceMemoryHeap[i].ui32Attribs |=
PVRSRV BACKINGSTORE LOCALMEM CONTIG;

				/*
					map the device memory allocator(s) onto
					the device memory heaps as required
				*/
				psDeviceMemoryHeap[i].psLocalDevMemArena = gpsSysData-
>apsLocalDevMemArena[0];

			}
			break;
		}
		default:
		break;
	}

	/* advance to next device */
	psDeviceNode = psDeviceNode->psNext;
}
```

The code fragment above finds the SGX ‘device node’ and changes the SGX heap configuration to use the new local backing store (UMA is the default). Non-UMA is selected by setting

- psDeviceMemoryHeap[i].psLocalDevMemArena to a valid backing store RA
- OR in PVRSRV_BACKINGSTORE_LOCALMEM_CONTIG into psDeviceMemoryHeap[i].ui32Attribs to change the heap attribute to be explicity non-UMA

Note: if the device has an MMU then the page tables will be allocated using the psDeviceNode->psLocalDevMemArena

When SysDeinitialise() is called all physical backing stores RAs should be destroyed, e.g.

```
	RA_Delete(gpsSysData->apsLocalDevMemArena[0]);
	gpsSysData->apsLocalDevMemArena[0] = IMG_NULL;
```

Appendix E. Single Threaded Kernel Services

Kernel Services is designed to be single threaded. Ports to Operating Systems that support multi-threaded, re-entrant kernels must ensure that kernel services remains single threaded irrespective of the 'flexibility' provided by the OS. Generally, single threaded execution is achieved by taking a mutex across all function interfaces to kernel services and includes:

- Ioctl bridge
- Power Management Interface

Any port to a new OS should implement the most optimal single threaded solution for that particular OS.

Multi-Processor (MP) Systems

Generally, hardware LISR handlers and deferred MISR operations should be able to occur in any driver context. In the case of MP systems the same is true but further protection may be required and the following must be guaranteed:

- A hardware LISR handler will not have the same interrupt handler called on any CPU until it has completed.
- A deferred MISR handler will not have the same MISR running on any CPU until it has completed.

Ports to Operating Systems that don't support this requirement must enforce mutual exclusion to achieve the consistency over shared data in all of these contexts of interference.

Appendix F. Cache Coherency Guidelines

SGX Services utilises memory with various caching attributes and special care must be taken when porting to a new OS and/or system. The OS kernel/system combination must guarantee correct behaviour when memory accesses to allocations with varying cache attributes interact. A typical example is an ‘L2 CPU cache’ and cases where an OS kernel may not enforce cache/memory coherency when sequencing accesses to the same memory with differing cache attributes. The correct behaviour expected of an OS kernel is to flush/invalidate the cache whenever any non-cached (uncached, write combined) memory is allocated and/or mapped.

Appendix G. Implementing Cache functions

Cache Flush functions OSFlushCPUCacheKM, OSCleanCPUCacheKM, OSInvalidateCPUCacheKM, OSFlushCPUCacheRangeKM, OSCleanCPUCacheRangeKM, OSInvalidateCPUCacheRangeKM should implemented when porting to a new platform. Some CPUs require explicit flush of different levels of caches (Example: ARM inner/outer caches, MIPS primary/secondary caches). Care should be taken to implement the cache functions such that all levels of caches and internal buffers are guaranteed to be flushed/cleaned/invalidated as required.

In the case of ARM processors, OSFlushCPUCacheKM calls outer_flush_all. An implementation for this should be added to the linux kernel, if the version of the kernel used does not already implement this.

Alternatively #define PVR_NO_FULL_CACHE_OPS to exclude call to outer_flush_all. This is safe to do, when outer cache is not present or when OSFlushCPUCacheKM is never used, which is the case when device memory and client surfaces are left uncacheable.

Appendix H. External System Cache Control Support

In some SOC designs there is a system cache which needs to be controlled (flush/invalidate) by the SGX microkernel but whose control register interface is not accessible to SGX. For this case there is an alternate build configuration which maps the cache control registers through the SGX MMU.

To enable this configuration define **SUPPORT_EXTERNAL_SYSTEM_CACHE** in the build.

It's also necessary to setup the following in the system specific code (sysconfig.h):

Device relative physical address of external cache control register.

```
#define SYS_EXT_SYS_CACHE_GBL_INV_REG_DEVPADDR      ?
```

This address is deconstructed into a 4KB page address and a page offset

```
#define SYS_EXT_SYS_CACHE_GBL_INV_REG_DEVP_PAGE_ADDR (SYS_EXT_SYS_CACHE_GBL_INV_REG_DEVPADDR &
0xFFFFF000)
#define SYS_EXT_SYS_CACHE_GBL_INV_REG_PAGE_OFFSET (SYS_EXT_SYS_CACHE_GBL_INV_REG_DEVPADDR &
0xFFF)
```

note: in some HW implementations the register written to is not actual the cache control register but another register, snooped by HW whose written value is re-written to the actual cache control register. From the point of the driver software and GPU operation both can be treated in the same way.

The register(s) is mapped into the SGX MMU at fixed virtual address (0x1000). It is expected that a single 4k page mapping is sufficient.

```
#define SGX EXT SYSTEM CACHE REGS DEVVADDR BASE 0x00001000
#define SGX_EXT_SYSTEM_CACHE_REGS_SIZE 0x00001000
```

SYS_EXT_SYS_CACHE_GBL_INV_REG_DEVP_PAGE_ADDR is mapped to device virtual address _EXT_SYSTEM_CACHE_REGS_DEVVADDR_BASE.

USSE assembler cache invalidate code macro applies the in page offset (SYS_EXT_SYS_CACHE_GBL_INV_REG_PAGE_OFFSET) to the access the register at the appropriate address.

USSE assembler cache invalidate macro (sample):

```
#define INVALIDATE_EXT_SYSTEM_CACHE_INLINE(RegA)                                          \
       wdf            drc0;                                                               \
       ldr            RegA, HSH(SGX_EXT_SYSTEM_CACHE_REGS_DEVVADDR_BASE), drc0;           \
       stad.bpcache   [RegA, HSH(SYS EXT SYS CACHE GBL INV REG PAGE OFFSET>>2)], HSH(VALUE);
       \
       idf            drc0, st;                                                           \
       wdf            drc0;
```

Where VALUE should be replaced with the appropriate value to be written to the cache control register.

Appendix I. Moving from Services 3.0 to Services 4.0

To make the transition from *Services 3.0* to *Services 4.0,* for a given operation system, it is necessary to adapt the system (SOC) component of the driver to support *Services 4.0.*

I.1. Porting the system (SOC) component from Services 3.0 to Services 4.0

With respect to the latest versions of *Services 3.0* and *Services 4.0,* the interface to the *system component* is unchanged: i.e. the prototypes for the system porting functions, declared in `syscommon.h`, are identical. (This was correct at the time of writing.) A minor change to the behaviour of `SysInitialise()` is required; this will be described in section I.3.

Few changes to the functionality of the *system porting functions* are necessary – a device capable of running the most up-to-date Services 3.0 driver should be able to run Services 4.0 after minimal porting of the system component.

System component code based on older versions of *Services 3.0,* however, will require a greater porting effort to support *Services 4.0.* It is assumed that your starting point is a fully up-to-date and functioning *Services 3.0* based driver.

For general information about system component architecture, refer to sections 4 and 0.

I.2. System component porting procedure

1. Validate the Services 3.0 system component by building and testing it with the Services 3.0 driver.
2. Install the *Services 4.0 DDK* in your build area; set environment variables as required.
3. Make the changes to the system code as described in section I.3.
4. Build the DDK – and expect build errors.
5. Syntax errors caused by missing or updated header files are the most likely errors. The modifications described in section I.3, should prevent many of these. Section I.3 is intended to be a guide only, and will not solve all build issues.
6. Be aware that (compared with *Services 3.0)* some header files may have been renamed or moved to different directories – *include paths* in system component build scripts may need to be updated.
7. You may find it useful to compare the *Services 4.0* version of one of the sample system component implementations (e.g. `Eurasia\services4\sgx_pc_i686_tc3`) with the corresponding system component implementation in *Services 3.0.*
8. Run the *unit tests* (`eurasia\unittests`) to validate the port to *Services 4.0.*

I.3. System component code modifications

This section is intended to be a general guide only. The recommended changes are with respect to generic example code and must be adapted to meet the requirements of specific systems. If these changes do not correspond to your system code, refer to the *Services 3.0* and *Services 4.0* sample implementations of `sgx_pc_i686_tc3`: the suggested modifications are based on the `sgx_pc_i686_tc3` system.

I.3.1. Changes to services4\system\<systemname>\sysconfig.c

Remove the following include statements:

```
#include "regpaths.h"
#include "sgxapi.h"
#include "pdump.h"
```

Add the following include statements:

```
#include "sgxinfo.h"
#include "pdump_km.h"
```

Add the following definitions to `sysconfig.c`; insert suitable values.

```
#define SYS_SGX_CLOCK_SPEED                 // insert system specific value
/* Defines for HW Recovery */
#define SYS_SGX_HWRECOVERY_TIMEOUT_FREQ     // insert system specific value
#define SYS_SGX_PDS_TIMER_FREQ              // insert system specific value
#define SYS_SGX_ACTIVE_POWER_LATENCY_MS     // insert system specific value
```

The Active Power Management latency can be modified per-system and 'tuned' accordingly. The latency is specified by the SYS_SGX_ACTIVE_POWER_LATENCY_MS define and is in millisecond units.

`SysInitialise()`

`SysInitialise()` initializes a timing information structure (`SGX_TIMING_INFORMATION* psTimingInformation`) with the constants described above. This data structure will be used by `services4\srvinit\devices\sgx\ta.asm`. Add the timing initialization code highlighted below to `SysInitialise()`.

```
PVRSRV_ERROR SysInitialise(IMG_VOID)
{
        IMG_UINT32              i;
        PVRSRV_ERROR            eError;
        PVRSRV_DEVICE_NODE      *psDeviceNode;
        SGX_TIMING_INFORMATION* psTimingInfo;

        gpsSysData = &gsSysData;
        OSMemSet(gpsSysData, 0, sizeof(SYS_DATA));

        gpsSysData->pvSysSpecificData = &gsSysSpecificData;
        gsSysSpecificData.ui32SysSpecificData = 0;

        eError = OSInitEnvData(&gpsSysData->pvEnvSpecificData);
        if (eError != PVRSRV_OK)
        {
                PVR_DPF((PVR_DBG_ERROR,"SysInitialise: Failed to setup env structure"));
                SysDeinitialise(gpsSysData);
                gpsSysData = IMG_NULL;
                return eError;
        }
        gsSysSpecificData.ui32SysSpecificData |= SYS_SPECIFIC_DATA_ENABLE_ENVDATA;

        /* Set up timing information*/
        psTimingInfo = &gsSGXDeviceMap.sTimingInfo;
        psTimingInfo->ui32CoreClockSpeed = SYS_SGX_CLOCK_SPEED;
        psTimingInfo->ui32HWRecoveryFreq = SYS_SGX_HWRECOVERY_TIMEOUT_FREQ;
        psTimingInfo->ui32ActivePowManLatencyms = SYS_SGX_ACTIVE_POWER_LATENCY_MS;
        psTimingInfo->ui32uKernelFreq = SYS_SGX_PDS_TIMER_FREQ;
```

I.3.2. Changes to services4\system\<systemname>\sysutils.c

Remove the following include statements:

```
#include "sgxapi.h"
#include "regpaths.h"
```

Add the following include statement:

```
#include "sgxinfo.h"
```

I.3.3. Changes to services4\system\<systemname>\sysconfig.h

Replace:

```
#define SYS_SGX_CLOCK_SPEED                 (100000000)

/* Defines for HW Recovery */
#define SYS_SGX_HWRECOVERY_TIMEOUT_FREQ     (100) // 10ms (100hz)
#define SYS_SGX_PDS_TIMER_FREQ              (500) // 2ms (500hz)
```

With the following definition; insert a suitable value:

```
#define SYS_SGX_USSE_COUNT                  // Insert suitable value here for the number of
                                            // USSE pipelines.
```

Remove the following definition from the *system specific configs* section:

```
/*****************************************************************************
 * system specific configs
 *****************************************************************************/

/* enable 'custom' synchronous HW recovery code for non-EDM enabled TestChip */
#define SGX_SYNCHRONOUS_TC1_HW_RECOVERY
```

Appendix J. PDUMP Limitations

J.1. PDUMP processes in standard single-app mode

The following processes are significant for PDUMP, listed in running order following driver start.

1. Kernel process: comments, allocate top-level Page Directory
2. pvrsrvinit / Window Manager Process: initialisation phase (plus optional window manager allocations)
3. client: everything else

J.2. PDUMP processes in multi-process mode

In server-client architectures such as X11, or composition environments such as Android/Compiz, workarounds we're previously required to generate valid PDUMPs. The driver can now be built with the SUPPORT_PDUMP_MULTI_PROCESS to enable this mode. Multiple Services clients can run simultaneously while parameters are only streamed for one target application under investigation.

The target of the PDUMP should be specified in the apphint configuration file in the application section with “PDumpActive=1”. At most one app should be marked for pdumping. The PDUMP feature relies on the PVR apphint API.

This feature is now enabled by default in Android DDKs.

Appendix K. Troubleshooting

Having completed the porting process if the driver appears to be non-functional there are various steps that should be carried out to ascertain the route cause:

1. check for SGX page faults by reading EUR_CR_BIF_INT_STAT
2. ensure Hardware recovery is not enabled in the build of the driver as this can mask the true cause of the problem
3. Check the host driver and USSE (microkernel) flags all match. Differences can result in mismatching structure definitions leading to unpredictable modes of failure
4. Verify modifications to the system files (sysconfig.c/h etc.). Incorrect modifications to these files are a common problem in the early stages of porting and debug.
5. Dump PDump parameters if possible/appropriate. Run these on the CSim and/or pass them to Imagination for further checking

If these steps fail to address the problem then contact Imagination for support