SGX Services Microkernel

Overview

Filename	:	SGX Services Microkernel.Overview.doc
Version	:	1.1.5 External Issue
Issue Date	:	08 Jun 2009
Author	:	PowerVR

Contents

1. Introduction

This document provides a general overview of the SGX microkernel within Services. The following items are covered:

- Microkernel overview
- Key features and responsibilities
- Microkernel scheduler
- Driver interface and associated data structures

Note: It is assumed that the reader has a basic knowledge of driver development.

2. Related documentation

Related Documents
SGX Services 4.Software Architectural Specification
SGX Services 4.Software Functional Specification
SGX Services Microkernel.Overview

3. Structure Naming Conventions

All Microkernel interface structures take the prefix `SGXMKIF_`, e.g. `SGXMKIF_CMDTA`. Private Microkernel structures take the prefix `SGXMK_`, e.g. `SGXMK_TA3D_CTL.`

Note: older versions of the DDK may have structures equivalent to `SGXMKIF_*` and `SGXMK_*` but take the prefix `PVR3DIF4_.`

4. Microkernel Overview

The SGX microkernel is a collection of micro-code routines that execute on the SGX USSE hardware component.

The microkernel services requests from the SGX driver and also handles hardware events from associated SGX hardware components. The microkernel driver interface consists of a single ‘kernel’ circular command buffer (CCB) into which the driver inserts command packets. These command packets are small, specifying the command type and referring to an associated ‘per-context’ command packet (via a hardware context structure). The ‘per-context’ command packets reside in Context CCBs (pointed to by hardware context structures) – these command packets contain the actual command data that SGX hardware works on. The driver ‘schedules’ the microkernel to process the command CCBs by writing to a single hardware register.

Execution of the USSE micro-code is triggered by the SGX PDS hardware component that schedules tasks on the USSE in response to incoming software and hardware events. Software events derive from the SGX driver, either submitting work items (renders, blits etc.) or simply signalling to the microkernel to process its internal work items (L/MISR events). Hardware events derive from SGX hardware components signalling completion (or some other condition) to the PDS.

The microkernel code paths to be executed depend on the type of event in the PDS. The PDS passes the event type to the USSE microkernel and the information is used to branch (or ‘loopback’) to the appropriate code handling path.

5. Microkernel Features

5.1. Key Advantages

The SGX microkernel is essentially a scheduler that services the requests of one or more applications that submit work items to SGX via the SGX driver. The primary responsibility of the microkernel is to schedule work items on the SGX hardware in the most efficient way. The microkernel provides key advantages over simple synchronous software scheduling and includes:

- Providing maximum performance by ensuring all hardware components are fully utilised, aiming for concurrent execution where possible
- Low latency (short interval between task complete to next task start)
- Offloading the CPU from hardware scheduling
- Effective CPU and SGX overlap
- Effective hardware resource sharing in multi-process environments with low latency requirements

5.2. Key Responsibilities

The SGX microkernel has a number of responsibilities and related features. The following sections summarise the important roles and features of the microkernel.

5.2.1. TA, 3D and Host CPU Overlap

The microkernel facilitates concurrent operation of the TA, 3D and Host CPU resulting in significant performance increases.

5.2.2. Memory address space management

SGX variants support between 1 and 16 concurrent SGX address spaces in the Bus Interface (BIF) hardware. Each SGX render context requires its own address space (one render context per application). The microkernel handles the allocation of these address spaces as well as eviction policy and associated cache flush controls.

5.2.3. Smart Parameter Management (SPM)

All PowerVR cores are deferred (or retained) mode render engines. A 3D scene’s geometry is converted by the TA into a display list which is a tiled representation of the scene. The 3D reads the display list and renders the scene tile-by-tile.

In SPM mode the TA runs out of space to accommodate the scene’s display list. In this event the 3D ‘partially’ renders the scene using the incomplete display list, the display list memory is freed and then the TA is resumed. Between each partial render the pixel depth information must be stored and loaded to and from an external depth buffer.

The microkernel handles all aspects of SPM.

5.2.4. Fast Hardware Recovery Support

It is possible for unexpected error conditions to occur leading to a lock-up in the hardware. To ensure that these conditions have minimal impact on the end user experience a hardware recovery mechanism is included that will detect any unexpected lockup and automatically recover the hardware and driver.

5.2.5. OS (Passive) Power Management Support

OS (Passive) Power Management is where the SGX driver responds to requests from the OS power manager to transition from one power state to another. When transitioning into a low power state the driver must first idle the SGX hardware before removing power from SGX. The microkernel provides a simple power management interface to the driver to idle SGX, signalling to the driver when complete.

Idling the SGX can occur in one of 3 ways depending on the power management response latency requirements:

- Flush all queued operations (current implementation)
- Flush all currently active operations
- Interrupt all currently active operations, storing out state to memory as required

5.2.6. Active Power Management Support

Active Power Management is where the SGX driver makes its own power management decisions. The microkernel code scheduled by PDS timer event looks for idle periods on the SGX hardware and powers down the core. The driver powers up the core when new hardware commands are submitted.

5.2.7. Resource Management Support

The resource manager is implemented in the software drivers but must ensure that hardware operations are flushed on all memory resources to be freed. The microkernel provides a simple resource management interface to the driver to flush operations on resources and signal to the driver when complete.

5.2.8. Occlusion Query Support

SGX 'ISP Breakpoints' implement occlusion queries. An ISP Breakpoint causes an interrupt in the microkernel which then updates control stream and visibility results in the client's buffer.

5.3. Synchronisation Objects

Synchronisation objects are used throughout the SGX driver and microkernel. The basic structure is defined below:

```
typedef struct _PVRSRV_SYNC_DATA_
{
        /* CPU accessible WriteOp Info */
        IMG_UINT32                          ui32WriteOpsPending;
        volatile IMG_UINT32                 ui32WriteOpsComplete;

        /* CPU accessible ReadOp Info */
        IMG_UINT32                          ui32ReadOpsPending;
        volatile IMG_UINT32                 ui32ReadOpsComplete;
} PVRSRV_SYNC_DATA;
```

'Sync objects' are used to track the submission order of read and write operations on a given resource and then ensure the order of processing is preserved when the read and write operations are scheduled asynchronously on the SGX hardware.

Note: there is the read and write operation distinction because reads may be (harmlessly) performed out of order with respect to two adjacent write operations that were submitted before and after a group of read operations.

Synchronisation objects are generally applied to memory buffers but can be attached to any resources that require synchronised access by hardware. Examples include render targets, textures and display surfaces.

Commands submitted to the microkernel may have any number of SRC and DST sync objects, where DST is a write operation and SRC is a read operation. Associating sync objects with a command results in dependency checking before commands can be executed. More specifically, at the point the command is queued in the driver, the driver 'snapshots' the 'pending' values of the sync objects and embeds these in the command along with the device virtual address for the sync object itself.

The microkernel will not process a given command until sync object 'complete' values match the snapshot pending values; when they do match the command can be started.

When a command completes the SRC sync objects read complete values are incremented and the DST sync objects write complete values are incremented

5.3.1. Special case applications

TA vs. 3D

A special sync object is created for every render context structure (psTA3DSyncObject).

This sync object is used when a TA operation is dependent on the result of an otherwise unrelated 3D operation. This sync object is enabled using the TA and Render flags

SGXMKIF_TAFLAGS_DEPENDENT_TA and SGXMKIF_RENDERFLAGS_TA_DEPENDENCY

Use case example: render to a vertex/index buffer vs. vertex/index buffer read by TA.

Transfer queue vs. TA

A special sync object is created for every transfer context structure (psTASyncObject).

This sync object is used when a transfer and TA operations are interdependent. This sync object is enabled by specifying valid device virtual addresses for sTATQSyncWriteOpsCompleteDevVAddr and sTATQSyncReadOpsCompleteDevVAddr.

Use case example: index/vertex buffer upload/blit/copy/fill by transfer queue vs. index/vertex buffer reads by TA.

Transfer queue vs. 3D

A special sync object is created for every transfer context structure (ps3DSyncObject).

This sync object is used when a transfer and 3D operations are interdependent. This sync object is enabled by specifying valid device virtual addresses for s3DTQSyncWriteOpsCompleteDevVAddr and s3DTQSyncReadOpsCompleteDevVAddr.

Use case example: texture updates by transfer queue vs. texture reads by 3D.

6. Driver to Microkernel Interface

The driver to microkernel interface consists primarily of memory structures mapped into the CPU and SGX address spaces (SGX mappings are done via its own MMU page tables). With the exception of the boot/initialisation phases, the services driver is largely decoupled from direct hardware control with the only hardware access being a single SGX register write to schedule the microkernel from the driver:

```
Register offset: EUR_CR_EVENT_KICK
Register value: EUR_CR_EVENT_KICK_NOW_MASK
```

6.1. Geometry processing and rasterisation

The primary interface to the microkernel is for scheduling geometry processing and rasterisation operations which are paired in a single command whose structure is defined by `SGXMKIF_CMDTA`.

`SGXMKIF_CMDTA` commands encapsulate all the information required for the SGX microkernel to schedule hardware geometry processing operations consisting of vertex shading and tiling on the USSE and TA (MTE/TE) respectively (see the core specific SGX TRM for details). The results are stored as a display list in memory.

The geometry processing and rasterisation interface allows for multiple geometry processing commands to be submitted without any associated rasterisation. Each new geometry processing operation is appended to a display list stored in memory. Flags in the command delimit batched sequences of geometry processing on a single display list; where `SGXMKIF_TAFLAGS_FIRSTKICK` starts processing to a new display list and `SGXMKIF_TAFLAGS_LASTKICK` terminates processing on the current display list.

When the TA terminates processing on a display list it generates an event in the microkernel which, if requested, queues the terminated display list for subsequent rasterisation.

6.1.1. Per-Render Context CCB

`SGXMKIF_CMDTA` commands are inserted into a ‘per-render context’ circular command buffer (CCB) by the services client driver. It is generally expected that each application will maintain a single per-context CCB of a given type, where types include transfer contexts, 2d hardware contexts and render contexts (as discussed here).

The microkernel accesses the per-context CCB via its parent render context, sTACCBBaseDevAddr and sTACCBCtlDevAddr), where the render context is embedded in the kernel CCB command, SGXMKIF_SGX_COMMAND.

6.2. Kernel CCB

In order for the microkernel to service a new command inserted into a per-context CCB an associated command must also be inserted to the central ‘kernel CCB’. The kernel CCB command structure is defined by SGXMKIF_SGX_COMMAND.

6.2.1. Services Driver Details

The per-context CCB structure is defined below:

```
typedef struct _SGX_CLIENT_CCB_
{
        /*!< meminfo for CCB in device accessible memory */
        PVRSRV_CLIENT_MEM_INFO        *psCCBClientMemInfo;
        /*!< meminfo for CCB control in device accessible memory */
        PVRSRV_CLIENT_MEM_INFO        *psCCBCtlClientMemInfo;
        /*!< linear address of the buffer */
        IMG_UINT32                    *pui32CCBLinAddr;
        /*!< device virtual address of the buffer */
        IMG_DEV_VIRTADDR              sCCBDevAddr;
        /*!< linear address of the write offset into array of commands */
        IMG_UINT32                    *pui32WriteOffset;
        /*!< linear address of the read offset into array of commands */
        volatile IMG_UINT32           *pui32ReadOffset;
        /*!< ((Size of the buffer) - (overrun size)) */
        IMG_UINT32                    ui32Size;
        /*!< Allocation granularity */
        IMG_UINT32                    ui32AllocGran;
}SGX_CLIENT_CCB;
```

The per-context CCB is created as part of the SGX render context, itself created by the API, `SGXCreateRenderContext`

```
PVRSRV_ERROR SGXCreateRenderContext(PVRSRV_DEV_DATA *psDevData,
                                    PSGXMKIF_CREATERENDERCONTEXT psCreateRenderContext,
                                    IMG_HANDLE *phRenderContext,
                                    PVRSRV_CLIENT_MEM_INFO **ppsVisTestResultClientMemInfo)
```

Space is acquired in the CCB by calling the API:

```
IMG_PVOID SGXAcquireCCB(SGX_CLIENT_CCB *psCCB, IMG_UINT32 ui32CmdSize)
```

The void pointer returned is used to write the command into the CCB.

Once the command is written into the CCB the CCB control 'write offset' is updated using the macro:

```
#define UPDATE_CCB_OFFSET(Off, PacketSize, CCBSize)
```

An associated command is also inserted into the kernel CCB whose structure is defined below:

```
typedef struct _SGX_CCB_INFO_
{
        /*!< meminfo for CCB in device accessible memory */
        PVRSRV_KERNEL_MEM_INFO        *psCCBMemInfo;
        /*!< meminfo for CCB control in device accessible memory */
        PVRSRV_KERNEL_MEM_INFO        *psCCBCtlMemInfo;
        /*!< linear address of the array of commands */
        SGXMKIF_COMMAND               *psCommands;
        /*!< linear address of the write offset into array of commands */
        IMG_UINT32                    *pui32WriteOffset;
        /*!< linear address of the read offset into array of commands */
        volatile IMG_UINT32           *pui32ReadOffset;
        /*!< lock for access to the CCB */
        PVRSRV_RESOURCE               sSGXScheduleCCBCommandResource;
#if defined(PDUMP)
        /*!< for pdumping */
        IMG_UINT32                    ui32CCBDumpWOff;
#endif
} SGX_CCB_INFO;
```

The kernel CCB commands are inserted into the CCB using the API:

```
PVRSRV_ERROR SGXScheduleCCBCommandKM (PVRSRV_SGXDEV_INFO           *psDevInfo,
                                      SGXMKIF_COMMAND_TYPE         eCommandType,
                                      SGXMKIF_COMMAND              *psCommandData,
                                      IMG_UINT32                   ui32CallerID)
```

It is at this point that the Services driver writes to the kick register EUR_CR_EVENT_KICK.

6.2.2. Context CCB (TA) Command Processing Details

Using raster commands as an example, having already incremented `ui32TACount` in the kernel CCB command's SGXMKIF_HWRENDERCONTEXT the microkernel reads the next SGXMKIF_CMDTA command packet.

Note: 'TA Count' is essentially a count of the queued TA commands for the render context.

SGXMK_TA3D_CTL is the central SGX microkernel control structure and contains two run lists: one of all SGXMKIF_HWRENDERCONTEXT structures that contain unfinished TA commands (geometry processing) and one of all SGXMKIF_HWRENDERCONTEXT structure that contain unfinished render commands (rasterisation).

Note: There is also a run list for Transfers

A new SGXMKIF_CMDTA's SGXMKIF_HWRENDERCONTEXT structure is added to the SGXMK_TA3D_CTL unfinished TA commands run list tail (sPartialRenderContextTail) if the TA Count value is 1.

Next, the microkernel checks to see if the TA is already active on another command. If not, the microkernel traverses the unfinished TA commands run list of SGXMKIF_HWRENDERCONTEXT. For the first SGXMKIF_CMDTA that the microkernel finds, the associated SGXMKIF_HWRENDERCONTEXT is promoted to the head of the unfinished TA commands run list of SGXMKIF_HWRENDERCONTEXT structures in SGXMK_TA3D_CTL. Finally, the TA is configured for the new command and then started.

6.2.3. Context CCB (TA) Command Completion Details

When the TA completes processing the current TA command an event is generated in the PDS and the microkernel is scheduled. One of two things happens to the command's SGXMKIF_HWRENDERCONTEXT depending on the value of the 'TA Count':

1. If there are more unfinished TA commands for this SGXMKIF_HWRENDERCONTEXT the structure will be moved from the head to the tail of the SGXMK_TA3D_CTL unfinished TA commands run list. This is done to 'round-robin' the TA between all applications' SGXMKIF_HWRENDERCONTEXT.
2. If there are no more unfinished TA commands for the HWRENDERCONTEXT, the structure will be removed from the SGXMK_TA3D_CTL unfinished TA commands run list.

With the SGXMKIF_HWRENDERCONTEXT structure dealt with, the TA command's SGXMKIF_HWRENDERDETAILS structure is handled next. Each SGXMKIF_HWRENDERCONTEXT has two run lists of SGXMKIF_HWRENDERDETAILS: one for unfinished TA commands and one for unfinished 3D commands.

3. If this is the first TA command but not the last TA command for the scene, the command's SGXMKIF_HWRENDERDETAILS structure is inserted into the unfinished TA commands run list within the HWRENDERCONTEXT structure.
4. If this is the last TA command for the scene (indicated by the SGXMKIF_TAFLAGS_LASTKICK flag) then the scene is ready for rendering. In this case the SGXMKIF_HWRENDERDETAILS structure is removed from the unfinished TA commands run list and added to the tail of the unfinished 3D commands run list within the HWRENDERCONTEXT structure.
5. If this kick was neither the first nor the last TA command of a scene then the SGXMKIF_HWRENDERDETAILS will be left on the unfinished TA commands run list within the HWRENDERCONTEXT.

As part of this sequence the microkernel can optionally write out status values to status addresses, both of which are specified by the TA command.

The microkernel checks for scenes ready to be rendered and then checks for other TA commands to process before exiting.

6.2.4. Render Processing Details

Following a TA complete processing, the microkernel checks for scenes that can be rendered. This is done be traversing the SGXMK_TA3D_CTL unfinished 3D commands run list of SGXMKIF_HWRENDERCONTEXT structures. Given a SGXMKIF_HWRENDERCONTEXT the microkernel traverses the unfinished 3D commands run list of HW render details. The first SGXMKIF_HWRENDERDETAILS structure that the microkernel finds will have its synchronisation state checked as a requirement of execution.

The synchronisation requirements are embedded in the original SGXMKIF_CMDTA and made up from two lists of pointers to synchronisation objects, one list for read operations and one list for write operations. Each synchronisation object pointer is partnered by a read and write operation dependency value – effectively the state of synchronisation object at the time the command was queued. These read and write operation values must match the actual synchronisation object values before a command can execute.

Note: Generally synchronisation objects are associated with memory buffers and are used to enforce the sequencing of asynchronous operations on the buffers, see the Services SFS and synchronisation section for more details.

Once the synchronisation check passes the SGXMKIF_HWRENDERDETAILS is used to configure the 3D for rendering. The associated SGXMKIF_HWRENDERCONTEXT is moved to the head of the SGXMK_TA3D_CTL unfinished 3D commands run list of SGXMKIF_HWRENDERCONTEXT structures and the 3D is started.

6.2.5. Render Completion Details

There are two types of render complete:

Mid-Scene Render Complete	The PIXELBE_EOR event is received and the scene's parameter memory is not freed by the DPM.
Normal Render Complete	The PIXELBE_EOR and 3DMEMFREE events and the scene's parameter memory is freed by the DPM

In either case, the SGXMKIF_HWRENDERDETAILS associated with the render is removed from the SGXMKIF_HWRENDERCONTEXT unfinished 3D commands run list. If there are more SGXMKIF_HWRENDERDETAILS in the unfinished 3D commands run list, the SGXMKIF_HWRENDERCONTEXT is moved to the tail of the SGXMK_TA3D_CTL unfinished 3D commands run list of SGXMKIF_HWRENDERCONTEXT structures. If there are not any more SGXMKIF_HWRENDERDETAILS in the unfinished 3D commands run list then the SGXMKIF_HWRENDERCONTEXT is removed from the unfinished 3D commands list.

The microkernel can optionally write out a set of 3D status values which are included in the SGXMKIF_HWRENDERDETAILS structure. The microkernel also updates the synchronisation objects on which the original command's execution depended.

Finally, the microkernel checks the unfinished TA and 3D command SGXMKIF_HWRENDERCONTEXT run lists for more TA and 3D commands to start before exiting.

Note: Mid-Scene Render is where the driver or application requires that an incomplete scene is rendered. More specifically, only part of the scene's geometry processing is completed and *that geometry rendered before resuming geometry processing for the rest of the scene. Mid-Scene Renders do not have their associated parameter memory freed on render complete. See SPM section for more details*

6.3. The Transfer Queue

The Transfer Queue interface provides a blitting API and includes commands of the following type:

- SRCCOPY
- COLOURFILL
- Presentation
- Texture upload
- Texture twiddle
- Mipmap generation

Transfer queue context and command structure are defined by SGXMKIF_HWTRANSFERCONTEXT and SGXMKIF_TRANSFERCMD, respectively.

6.3.1. Services Driver Details

Transfer commands are inserted into a transfer context CCB, created as part of the transfer context which itself is created by a call to the function SGXCreateTransferContext.

```
PVRSRV_ERROR SGXCreateTransferContext(PVRSRV_DEV_DATA *psDevData,
                                      IMG_UINT32 ui32RequestedSBSize,
                                      SGXMKIF_TRANSFERCONTEXTCREATE *psCreateTransfer,
                                      IMG_HANDLE *phTransferContext)
```

The transfer context CCB structure is the same as the geometry processing and rasterisation (`SGX_CLIENT_CCB`). Similarly, each new transfer command is also paired with a command in the kernel CCB where the kernel CCB command type (eCmdType) is set to SGX_KERNEL_CCB_TRANSFER_CMD.

6.4. MISR Microkernel Scheduling

Operations on external devices such as Vertical Synchronisation (VSync) flips on display hardware interact with SGX microkernel processing and potentially block transfer and rendering operations. The MISR microkernel scheduling interface allows external devices to signal asynchronously to the microkernel that external device operations have completed and to perform SGX command processing.

Like geometry processing and rasterisation commands, the microkernel scheduling commands are also inserted into the kernel CCB but the command type (SGXMKIF_SGX_COMMAND.eCmdType) is set to SGX_KERNEL_CCB_PROCESSQ_CMD. Note: the render context address union member does not need to be setup for this type of command.

6.5. OS (Passive) Power Management interface

OS (Passive) power management is where the SGX driver receives power management requests from the OS power manager. The passive power management microkernel interface is simple and consists of a single data member, `ui32PowManFlags`, in the main microkernel control structure, defined by SGXMKIF_HOST_CTL.

When the OS power manager requests a power-down transition the Services driver adds the value `SGXMKIF_POWMAN_POWEROFF_REQUEST` into `ui32PowManFlags`. This signals to the microkernel that a power down is pending and the hardware must be idle. When the microkernel successfully brings the hardware to the idle state it adds the value `SGXMKIF_POWMAN_POWEROFF_COMPLETE` into `ui32PowManFlags` which the driver polls on before it proceeds with power down of the chip.

6.6. Active Power Management interface

Active power management is where the microkernel internally detects periods of inactivity and powers down the chip until the next operation is submitted. The microkernel's internal timer task is used to identify periods of inactivity and generates an external interrupt to power down SGX from the Services MISR. The MISR calls the Services set power state function, `PVRSRVSetDevicePowerStateKM`, to power down SGX:

New operations submitted to SGX while actively powered down are preceded by a call to the set power state function `PVRSRVSetDevicePowerStateKM` to ensure power is returned to the device before resuming hardware operations.

6.7. Resource Management interface

Services Resource Management ensures all resources associated with a process are freed after a process expires, cleanly or otherwise. The microkernel provides a resource management interface that allows the driver to request that the hardware flushes all operations on resources associated with the expired process. The interface is similar to the passive power management where the driver sets the value, `SGXMKIF_RESMAN_CLEANUP_REQUEST,` in the flags member `ui32ResManFlags` in the `SGXMKIF_HOST_CTL` structure. However, in the case of resource management, the driver also specifies the hardware render context associated with the expired process by setting-up the member, `sResManRenderContext`, also in `SGXMIF_HOST_CTL.`

The driver polls on `ui32ResManFlags` for the microkernel to set the bit, `SGXMKIF_RESMAN_CLEANUP_COMPLETE`. When the microkernel timer detects the clean-up request it ensures the old hardware render context is no longer accessed by hardware before setting the bit `SGXMKIF_RESMAN_CLEANUP_COMPLETE in ui32ResManFlags,` causing the driver poll to succeed.

The driver resource management clean-up code can now proceed with the actual clean-up operations.

6.8. Hardware Recovery interface

Hardware Recovery is where the microkernel and/or driver detects that one of more hardware operations have failed and takes appropriate action, recovering the hardware and then processing the failed hardware operations.

Hardware recovery detection takes place generally on the microkernel but there is also support in the driver for independent hardware recovery detection in the case of the microkernel itself being affected by a failed hardware operation.

6.8.1. Microkernel Detection

The microkernel uses the timer event to periodically check hardware status and in the event of detecting a problem will generate an external interrupt, scheduling the LISR and then MISR. The microkernel sets the bit `SGXMKIF_INTERRUPT_HWR` in `ui32InterruptFlags` in the `SGXMKIF_HOST_CTL` structure, signalling to the MISR that hardware recovery is required. The MISR then calls the function, `HWRecoveryResetSGX`, to reset the SGX hardware, re-enable the microkernel timer and reschedules the micro-kernel.

6.8.2. Host Driver Detection

The Services driver hardware recovery detection does not rely on the microkernel and uses an OS timer installed at initialisation (see the Services function `OSAddTimer` in the DDK source).

The OS timer periodically calls the Services function, `SGXOSTimer`, which checks the number Event Data Master (EDM) tasks serviced on all USSE pipes against the number recorded on the previous call to `SGXOSTimer.` If the two counts are the same then hardware recovery is required and `HWRecoveryResetSGX` is called in the same way as microkernel detection.

6.9. Microkernel and ISR Interactions

In the general case the SGX microkernel can be considered to be a self-contained, autonomous entity that receives commands from the Services driver and schedules commands asynchronously and in the most optimal fashion and where no driver intervention is required after the commands have been submitted. However, there are exceptions where the SGX microkernel must generate external interrupts on the Host CPU or external device ISRs must reschedule (kick) the SGX microkernel in response to a change in synchronisation state: Examples include:

- External device ISRs (display and buffer devices). In this case SGX operations may be blocked waiting for external device operations to complete (e.g. Vsync flip) and when the operation does complete the external device ISR must reschedule the microkernel. Similarly, external device operations may be blocked by SGX operations (e.g. Render and Blit) and so the microkernel must generate an external interrupt to schedule the SGX ISR to start external device operations.
- Active power management. When the microkernel detects that SGX has been idle for a certain period of time it can power down SGX. It does this by generating an external interrupt to schedule the SGX LISR from where SGX is actually powered down. The subsequent 'wake-up' of SGX is triggered by the insertion of a new command in the CCBs.
- Microkernel Detected Hardware recovery. In this case the microkernel must schedule the LISR in order to reset SGX.
- OS Event Objects. OS event object may be used to improve system power management savings and multi-process performance. When the host driver waits for a resource to become available or an operation to complete, the waiting process can de-schedule and be woken-up when an event object is signalled. Events in the microkernel (TA, 3D, blit complete) correspond to potential wake-up events for de-scheduled processes so the microkernel must schedule the SGX ISR in order to signal the OS event object(s).