SGX Services Microkernel

Scheduling

Filename	:	SGX Services Microkernel.Scheduling.doc
Version	:	1.0.23 External Issue
Issue Date	:	31 Jul 2009
Author	:	Imagination Technologies

Contents

1. Introduction

This document provides descriptions of the various scheduling features of the SGX microkernel within Services. The following items are covered:

- Scheduling overview
- Methods available
- Memory requirements and detailed description of the hardware control procedure.

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. Terminology

The SGX driver supports two modes of scheduling: ‘basic scheduling’ and ‘low latency scheduling’ (LLS), where the mode selected depends on the compile option `SGX_SUPPORT_LOW_LATENCY_SCHEDULING` (not setting it selects basic scheduling). This section provides a brief overview of each of the modes.

4.1. Basic Scheduling

This is when each command packet is allowed to run to completion before the next command can be started. This is the mode of operation used when Low Latency Scheduling support is not compiled into the driver.

4.2. Low Latency Scheduling

Low Latency scheduling allows for a finer level of control on command packets to be executed by the SGX Microkernel. This finer level of control has the advantage of reducing latency times for stopping current commands and starting other commands.

When the compile option `SGX_SUPPORT_LOW_LATENCY_SCHEDULING` is defined the actual method used to support Low Latency Scheduling depends on the version/revision of the SGX hardware.

4.2.1. Software Scheduling (Rev 1)

This mode of operation is used when the SGX hardware is not capable of providing direct support in the hardware. In this mode the all client APIs attempt to subdivide any given scene into at least 4 TA command packets (4 TA kicks per scene).

For the 3D phase the render is broken down into Macro-tile sized sub-renders, allowing other renders of higher priority contexts to be started.

4.2.2. Hardware Supported Context Switching (Rev 2)

This mode of operation is used when the SGX hardware is capable of providing support for interruption of TA and 3D phases. In this mode TA command processing can be interrupted at the per-index level. When the TA phase is interrupted the SGX Microkernel samples several SGX status registers to determine what action is required for the command to be resumed. When the hardware completes the context switch the secondary attribute state of the USSE is stored to memory and once the TA has finished it is instructed to store the MTE state to memory. The resume phase is the opposite of this: the MTE state is restored, the secondary attributes loaded and the TA is then restarted.

Like the TA phase, the 3D phase render can be interrupted at a low level by the hardware (per-triangle, per-tile level). Like TA interruption, when the SGX hardware completes the interruption of the render the SGX Microkernel samples several status registers to determine the actions required for the render to be resumed later.

For both the TA and 3D phases the SGX hardware flushes internal state out to several driver allocated memory regions at the point of context switch. The advantage of this mode over the software scheduling mode is that renders are only interrupted as required.

4.2.3. Core Support

The DDK header file, sgxfeaturedefs.h, determines which scheduling mode a particular core version/revision implements where

`SUPPORT_SGX_LOW_LATENCY_SCHEDULING` is the low latency scheduling compile flag

`SGX_FEATURE_SW_ISP_CONTEXT_SWITCH` is Software Scheduling (macro tile rendering)

`SGX_FEATURE_VDM_CONTEXT_SWITCH_REV_2` is hardware VDM (TA) context switching

`SGX_FEATURE_ISP_CONTEXT_SWITCH_REV_2` is hardware ISP (3D) context switching

5. Scheduling Overview

The SGX Microkernel is responsible for all aspects of the scheduling of SGX and determines which command should be processed next based on complex state and command dependencies. Command dependencies are checked when either a new command is submitted from the host driver or a currently executing command has completed in the TA or 3D phase. There are two main types of dependency which are both checked before a command is chosen.

Context Priority	There is a three level priority scheme implemented in the driver: High, Medium and Low. The SGX Microkernel will always attempt to run commands from the higher priority contexts first before considering the medium and then finally low priority contexts. However, all commands are still subject to their own command dependencies which can result in context priority level rules being overridden.
Command Dependencies	Each command that is submitted to the SGX Microkernel may have one or more dependencies. These dependencies are generally synchronisation objects and are used to guarantee that commands submitted for operations on some resource (e.g. render target pixel buffer) are completed in the order they were submitted. These command dependencies can result in scheduling based on priority being overridden.

In addition to the dependencies above there is a simple rule which is always adhered to by the SGX Microkernel: for any given SGX Render Context every command submitted must be executed in submission sequence. In the TA phase this is enforced through the use of circular command buffers and in the 3D phase through design of the microkernel code.

6. Implementation Details

This section will cover the details of how each of the scheduling modes are implemented in the SGX Microkernel and where applicable the Client API.

6.1. Software Scheduling

As mentioned in section 4.2.1 this mode of operation requires no hardware support in SGX. This mode aims to reduce latency by splitting up TA commands and renders into smaller packets of work so processing can be switched from one context to another between packets. On the TA phase this is done by forcing the submission of TA command packets based on heuristics. The heuristics currently implemented in the driver work in one of two modes and both aim to achieve the same result which is to send 3 or 4 TA command packets per scene. The client API driver does this by tracking either the average number of primitives or indices sent on the previous scenes and then dividing this number by 4. When this calculated threshold is reached a TA command packet is submitted to the SGX Microkernel. However, the number of TA command packets is still limited by the size of the client API buffers, such as fragment shaders etc. This means that it is possible to have far more than 4 TA commands per scene.

For the 3D phase all the work required to subdivide the scene into macro-tile sub-renders is done by the SGX Microkernel. It does this by modifying the regions header for the last tile in each macro-tile to include the `LASTRGN` bit. This is done with the use of a lookup table which is calculated by the host services code at the point the render target was added. This lookup table contains 2 important pieces of information for the microkernel to use: first, the SGX device virtual address of the first region header word for the last tile in each macro tile and second, the use of the LSB in the address to indicate whether the macro tile is actually going to be rendered - this is important to cut down the amount of modifications the microkernel is required to make and to stop the microkernel attempting to kick a render on un-initialised regions headers which would cause a hardware lockup. An example of not all macro tiles being used is a small render target (16x16) where theoretically only a single tile is required. However due to the implementation of SGX there is a minimum of 4 macro tiles and each of these have a granularity of 4 x 4 tiles. This means that 3 out of the 4 macro tiles do not ever get used and therefore the region headers do not need patching. The LASTRGN bit is used to stop the ISP fetching any further tiles and when all the tiles up to this point have been completed the `PIXELBE_ENDRENDER` event is generated as if it were the completion of the whole render. When each of the macro tile renders has completed the microkernel then checks which macro tile is next and whether it is valid (using the lookup table). If the macro tile which has just been rendered is not the last, the microkernel will iterate through the table to find the next valid macro tile. Upon finding the next valid macro tile it will programme the `EUR_CR_ISP_RGN_BASE` register appropriately and restart the render. This sequence continues until the macro tile marked as the last macro tile has been rendered. At this point the render is treated as being completely finished and it is therefore removed from the microkernel run list and all the sync objects and status values are updated.

In addition to managing the adjustment of the region headers the microkernel is also required to make modifications to the DPM de-allocation settings. This is required to ensure that the parameters allocated to the global list are not de-allocated until the last macro tile for the render has completed. This is done by ensuring that if de-allocation of the parameters is required that the `EUR_CR_DPM_3D_DEALLOCATE_GLOBAL_MASK` is not set for any of the macro tile renders other than the last one. If the driver has requested that the parameters not be de-allocated this is still honoured.

6.2. Hardware Support Context Switching Rev 2

6.2.1. VDM Context Switching

Memory Requirements

Since this mode of operation uses features in the hardware memory blocks must be allocated in order for SGX to store its internal state at the point of VDM context store and then restore this stored data at the point of resuming the TA. The following table lists the memory blocks that are required for this mode of VDM context switching, memory block sizes and descriptions of what they are used for.

Name	Size (bytes)	Description

Name	Size (bytes)	Description
psVDMSnapShotBufferMemInfo	124 * sizeof(IMG_UINT32)	This buffer is used by the VDM modules within SGX to save its internal state. One of these is allocated per SGXMKIF_HWRENDERCONTEXT.
psMTEStateFlushMemInfo	(EURASIA_TACTL_ALL_SIZEINDWORDS + 1) * sizeof(IMG_UINT32)	This buffer is used to store a full set of MTE state. One of these is allocated per SGXMKIF_HWRENDERCONTEXT.
psVDMSAMemInfo New and 543/545 Specific. Replaces Client driver SA update program tracking.	512 * sizeof(IMG_UINT32)	This buffer is used to save all the secondary attributes. One of these is allocated per SGXMKIF_HWRENDERCONTEXT.
psContextSwitchPDSClientMemInfo	sizeof (g_pui32PDSVDMCONTEXT_STORE)	This memory is allocated for the PDS program used to send the context switch MTE state terminate. One of these is allocated at driver load time.
psContextSwitchUSEClientMemInfo	sizeof (pbUSEProgramVDMContextStore)	This memory is allocated for the USSE program used to send the context switch MTE state terminate. One of these is allocated at driver load time.
psUSETASARestoreMemInfo New and 543/545 Specific. Replaces Client driver SA update program tracking.	sizeof(pbUSEProgramTASARestore)	This memory is allocated for the USSE program used to restore the Secondary Attribute state. One of these is allocated at driver load time.
psPDSTASARestoreMemInfo New and 543/545 Specific. Replaces Client driver SA update program tracking.	sizeof(g_pui32PDSTASARESTORE)	This memory is allocated for the PDS program used to restore the Secondary Attribute state. One of these is allocated at driver load time.

Name	Size (bytes)	Description
psUSETAStateMemInfo	sizeof(pbUSEProgramTAStateRestore)	This memory is allocated for the USSE program used to restore the MTE state. One of these is allocated at driver load time.
psPDSTAStateMemInfo	sizeof(g_pui32PDSTASTATERESTORE)	This memory is allocated for the PDS program used to restore the MTE state. One of these is allocated at driver load time.

Context Store

When appropriate the microkernel is capable of requesting the VDM and thus TA to stop processing on a per-index level. The following gives the sequence of what happens when a context store is required.

1. The microkernel checks its internal flags to ensure that a context store is not already in progress.
2. The Microkernel will write the Device Virtual Address of the psVDMSnapShotBufferMemInfo to the EUR_CR_MASTER_VDM_CONTEXT_STORE_SNAPSHOT register.
3. The Microkernel will write the values required to run the context switch state terminate program into the EUR_CR_MASTER_VDM_CONTEXT_STORE_STATE0 and EUR_CR_MASTER_VDM_CONTEXT_STORE_STATE1 registers. These values are patched into the Microkernel code at driver load time using the STACS_TermState* label offsets. This program is also responsible for saving the state of any secondary attributes which may be allocated to memory.
4. The Microkernel the writes EUR_CR_MASTER_VDM_CONTEXT_STORE_START_PULSE_MASK to the EUR_CR_MASTER_VDM_CONTEXT_STORE_START. This action causes the VDM to run the context switch state terminate program, which results in the TA stopping the processing of any more indices and the VDM storing its internal state to the memory pointed at by the EUR_CR_MASTER_VDM_CONTEXT_STORE_SNAPSHOT register.
5. The Microkernel then polls on the EUR_CR_MASTER_VDM_CONTEXT_STORE_STATUS register.
    - 5.1. Firstly it polls for any of the bits within the register to be set. This indicates that the TA has started any operations required within the TA to carry out the context store.
    - 5.2. Next the Microkernel polls for the PROCESS bit to be 0. This indicates that all the operations required within the TA have completed.
6. Next the Microkernel checks to see which of the remaining status bits are set within the EUR_CR_MASTER_VDM_CONTEXT_STORE_STATUS register. Depending on the combination of bits remaining the Microkernel takes different paths.
    - 6.1. If only the NA bit is set this indicates that the VDM had not yet started processing the first index block. This means that Microkernel is not required to save any HW state. The Microkernel can then either start the higher priority TA or service the power request without any further work required for the context store.
    - 6.2. If both the NA and COMPLETE bits are both set this indicates that no context switch was performed as the VDM had reached the end of the control stream. This causes the Microkernel to wait for the TA finished event and treat it as a normal.
    - 6.3. If only the COMPLETE bit is set this indicates that the VDM has carried out a context store. In this case the microkernel will exit and wait for the TA finished event.
7. In the case where only the COMPLETE bit is set, upon receiving the TA_FINISHED event the Microkernel will handle the remaining operations required to be able to restore the HW state when the TA operation is resumed. The first of these is to save the EUR_CR_TE_STATE register value from each core to memory.

8. The Microkernel then moves the current SGXMKIF_RENDERCONTEXT to the tail of the sPartialRenderContext run list.
9. Next the Tail Pointer Cache is flushed to memory to ensure that the data is not written to the wrong memory at a later point. This is done by writing the EUR_CR_TE_TPCCONTROL_FLUSH_MASK value to the EUR_CR_TE_TPCCONTROL register. The Microkernel will wait for this operation to complete by polling on the EUR_CR_EVENT_STATUS register for the TPC_FLUSH bit to be set.
10. The Microkernel will then set the SGXMKIF_TAFLAGS_VDM_CTX_SWITCH in the ui32Flags member within the SGXMKIF_CMDTA to indicate that this command has been context switched. This is so the Microkernel knows that the restore steps need to be done when it comes back to processing this command.
11. The Microkernel then programs the EUR_CR_MTE_STATE_FLUSH_BASE register with the Device Virtual Address of psMTEStateFlushMemInfo. The action of writing the EUR_CR_MTE_STATE_FLUSH_MASK to the EUR_CR_MTE_STATE register causes the MTE to write it internal state to memory.
12. The Microkernel then polls on the EUR_CR_EVENT_STATUS2 register for the MTE_STATE_FLUSHED bit to be set. This indicates that the MTE has finished writing its internal state to memory.
13. Once all of these steps have been completed the Microkernel is able to process the action which required the context store to take place, whether this is a TA of a higher priority or a power down request.

Context Resume

When required the microkernel will do the following steps to setup the TAs internal state so it can resume a previously stored TA command.

1. Having written all the SGX registers associated with the TA to the values contained within the SGXMKIF_CMDTA. The Microkernel will check whether the command needs to be resumed by checking the SGXMKIF_TAFLAGS_VDM_CTX_SWITCH flag in the ui32Flags member of the command.
2. The Microkernel will write the Device Virtual Address of the psVDMSnapShotBufferMemInfo to the EUR_CR_MASTER_VDM_CONTEXT_STORE_SNAPSHOT register.
3. The Microkernel the writes EUR_CR_MASTER_VDM_CONTEXT_LOAD_START_PULSE_MASK to the EUR_CR_MASTER_VDM_CONTEXT_LOAD_START. This action causes the VDM to restore it internal state using the Memory pointed at by the EUR_CR_MASTER_VDM_CONTEXT_STORE_SNAPSHOT register.
4. The Microkernel then polls on the EUR_CR_EVENT_STATUS2 register for the VDM_CONTEXT_LOAD bit to be set. This ensures that the VDM has completed the loading of its state.
5. The Microkernel's next step is to restore the MTE state. This is done by writing the values required to the EUR_CR_MASTER_VDM_CONTEXT_STORE_STATE0 and EUR_CR_MASTER_VDM_CONTEXT_STORE_STATE1 registers. The values used are patched into the Microkernel code at driver load time using the RTA_MTEStateRestore* label offsets. The data in these values contain the setup information required to run the PDS (psPDSTAStateMemInfo) and USSE (psUSETAStateMemInfo) programs used to restore the MTE's internal state.
6. In order for the program to be sent to the PDS and run, the Microkernel writes the EUR_CR_VDM_TASK_KICK_PULSE_MASK to the EUR_CR_VDM_TASK_KICK.
7. Since the interface for kicking a task is only single buffered the Microkernel polls on the EUR_CR_EVENT_STATUS2 register for the VDM_TASK_KICKED bit to be set. This indicates the VDM has sent the task to the PDS and is now ready to receive another task if required.
8. Now that the VDM and MTE state have been restored the Microkernel will then restore any secondary attribute state that was saved. This is done in an identical manner to the MTE state, using the VDM_TASK_KICK interface. However this time it writes the values at the RTA_SARestore* offsets which are patched into the uKernel code at driver load. The data at

these offsets contain the setup information required to run the PDS (psPDSTASAMemInfo) and USSE (psUSETASAMemInfo) programs.

9. The Microkernel is now able to continue setup up of its own internal state before the TA command is restarted by the write to of EUR_CR_VDM_START_PULSE_MASK to the EUR_CR_VDM_START register.

6.2.2. ISP Context Switching

Memory Requirements

Since this mode of operation uses features in the hardware, memory blocks must be allocated in order for SGX to store its internal state at the point of context store and then restore this stored data at the point of resuming the 3D. The following table lists the memory blocks which are required for this mode of ISP context switching, memory block sizes and descriptions of what they are used for.

Name	Size (bytes)	Description
psZLSCtxSwitchMemInfo	EURASIA_ISPREGION_SIZEX * EURASIA_ISPREGION_SIZEY * 5	This buffer is used by the ZLS module within SGX to save the Z, Stencil and Mask plane information for the tile which the context store occurred on. One of these is allocated per SGXMKIF_RTDATA.
apsPDSCtxSwitchMemInfo	4096 * SGX_FEATURE_NUM_PDS_PIPES	This buffer is used by the PDS module to save state about primitives in flight. One of these is allocated per SGXMKIF_RTDATA.

Context Store

When appropriate the microkernel is capable of requesting the ISP to stop processing on the per-primitive level. The following gives the sequence of what happens when a context store is required.

1. The microkernel checks its internal flags to ensure that a context store is not already in progress.
2. The Microkernel will write the Device Virtual Address of the psZLSCtxSwitchMemInfo to the EUR_CR_ISP2_CONTEXT_SWITCH register.
3. The Microkernel will write the Device Virtual addresses of the apsPDSCtxSwitchMemInfo memory regions to the EUR_CR_PDS*_CONTEXT_STORE registers.
4. The Microkernel will then write EUR_CR_ISP_3DCONTEXT_STORE_MASK to the EUR_CR_ISP_3DCONTEXT register. This is the action which signals to the ISP that a context store is requested.
5. The Microkernel task will then exit and wait for the PIXELBE_EOR event.
6. When the PIXELBE_EOR event is received by the Microkernel, the Microkernel will check the EUR_CR_PDS register. If neither of the PDS_*_STATUS_CSWITCH_COMPLETED bits are set it indicates that no context store was required and the PIXELBE_EOR event was generated because the render had truly finished. This will result in the Microkernel treating the event like a standard end of render event and writes back any status values and removes the render from its run lists.
7. If any PDS_*_STATUS_CSWITCH_COMPLETED bits are set in the register, the SGX hardware was able to interrupt the render. In this case the Microkernel needs to do a few more actions so that the render can be correctly resumed at a later point in time.
    7.1. The Microkernel saves the value of the EUR_CR_ISP_STATUS2 register to memory. This is saved within the ui32NextRgnBase member of the SGXMKIF_HWRENDERDETAILS.
    7.2. The Microkernel then reads the EUR_CR_ISP_STATUS3 and EUR_CR_ISP_STATUS1 registers. It then combines both of these registers and saves the value within the

ui32ISPRestart member of the SGXMKIF_HWRENDERDETAILS. The registers are combined in the format required for the EUR_CR_ISP_START register.

7.3. The Microkernel then saves value of the EUR_CR_ISP2_CONTEXT_SWITCH3 registers to memory. This is saved within the ui32ISP2CtxResume3 member of the SGXMKIF_HWRENDERDETAILS.

7.4. The Microkernel then reads the EUR_CR_ISP2_CONTEXT_SWITCH2 register and then either sets or clears the PT_IN_FLIGHT* bits. The setting/clearing of these bits is based on the *_STATUS_PTOFF_STORED bits which are read from the EUR_CR_PDS. Once the value has been modified the value is saved to memory within the ui32ISP2CtxResume2 member of the SGXMKIF_HWRENDERDETAILS.

7.5. The last piece of hardware state the Microkernel is required to save is the EUR_CR_PDS*_CONTEXT_STORE registers. These registers are read and the values modified before being saved to memory. The values are modified by either setting or clearing the EUR_CR_PDS*_CONTEXT_RESUME_VALID_STREAM bits for both register values. The setting/clearing of these bits is based on the *_STATUS_CSWITCH_COMPLETED bits in the EUR_CR_PDS register. The values are saved within the asPDSStateBaseDevAddr of the SGXMKIF_HWRTDATA.

7.6. The Microkernel will then set the SGXMKIF_HWRTDATA_RT_STATUS_ISPCTXSWITCH in the memory pointed at by sRTStatusDevAddr. This is used to indicate that the render was context store and therefore needs to be resumed at a later time.

8. Once all of these steps have been completed the Microkernel is able to process the action which required the context store to take place, whether this is a render of a higher priority or a power down request.

Context Resume

When required the microkernel will carry out the following steps to setup the ISPs internal state so it can resume a previously stored render.

1. Having written all the SGX registers associated with the render to the values contained with the SGXMKIF_HWRENDERDETAILS, the Microkernel will check whether the command needs to be resumed by checking the SGXMKIF_HWRTDATA_RT_STATUS_ISPCTXSWITCH flag in the memory pointed at by sRTStatusDevAddr.
2. The Microkernel will write the values in the asPDSStateBaseDevAddr to the EUR_CR_PDS*_CONTEXT_RESUME registers.
3. The Microkernel then writes the value in the sZLSCtxSwitchBaseDevAddr to the EUR_CR_ISP2_CONTEXT_RESUME register.
4. The Microkernel then writes the value in the ui32NextRgnBase to the EUR_CR_ISP_RGN_BASE register.
5. The Microkernel then writes the value in the ui32ISPRestart to the EUR_CR_ISP_START register.
6. The Microkernel then writes the value in the ui32ISP2CtxResume2 to the EUR_CR_ISP2_CONTEXT_RESUME2 register.
7. The Microkernel then writes the value in the ui32ISP2CtxResume3 to the EUR_CR_ISP2_CONTEXT_RESUME3 register.
8. The Microkernel then writes EUR_CR_ISP_RENDER_TYPE_NORMAL3D_RESUME to the EUR_CR_ISP_RENDER register.
9. The Microkernel then continues with its own internal state setup before the render is restarted by the write of EUR_CR_ISP_START_RENDER_PULSE_MASK to the EUR_CR_ISP_START_RENDER register.