Vivante Graphics Driver Porting Guide

Overall architecture

Figure 1 lists the overall architecture of Vivante graphics software stack.

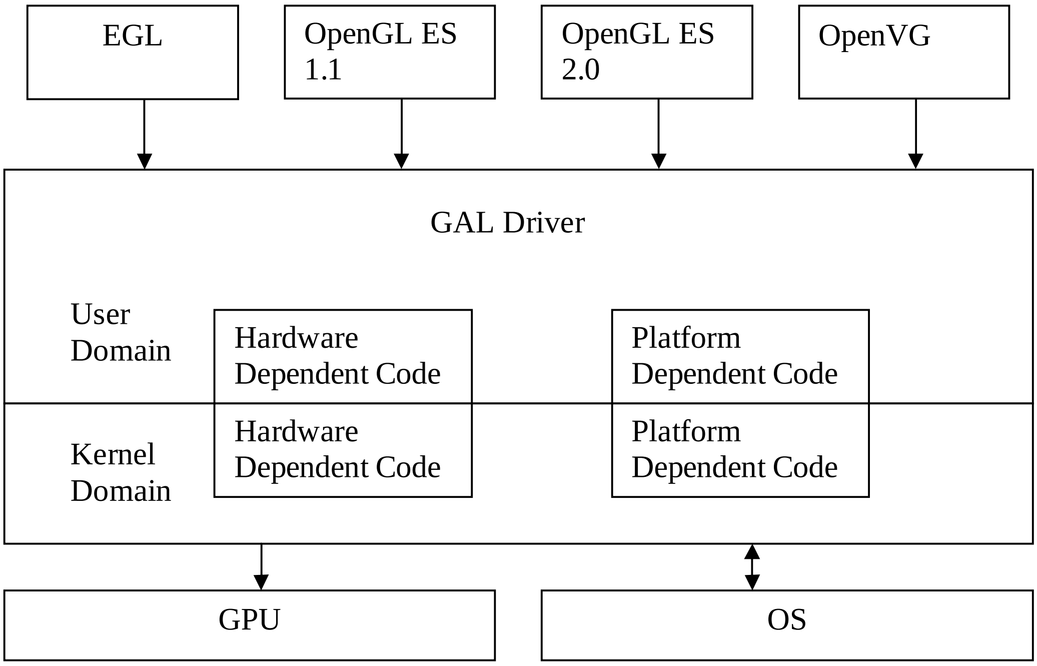

Figure 1 – Software Stack Overview

As porting is concerned, the GAL platform dependent code and EGL module are the major items.

Build the driver

The software modules are listed below:

Driver name	Type	Description	Source code directory
galcore	Kernel mode driver	GAL kernel mode driver	projects/hal/os/<OS >kernel projects/hal/kernel projects/arch/<ARCH>/hal/kernel
libGAL	Dynamic library	GAL user mode driver	projects/hal/os/<OS>user projects/hal/user projects/arch/<ARCH>/hal/user
libGLES_CM	Dynamic library	OES 1.1 (Common)	projects/drivers/openGL/es11
libGLES_CL	Dynamic library	OES 1.1 (Common Lite)	projects/drivers/openGL/es11
libGLESv2x	Dynamic	OES 2.0	projects/drivers/openGL/libGLESv2x

	library		
libOpenVG	Dynamic library	OVG 1.1	projects/drivers/openVG/vg11
libEGL	Dynamic library	EGL 1.3	projects/drivers/openGL/egl

Table 1 – Source code structure

Kernel driver loading

When the kernel driver is loaded, some initialization routines must be performed.

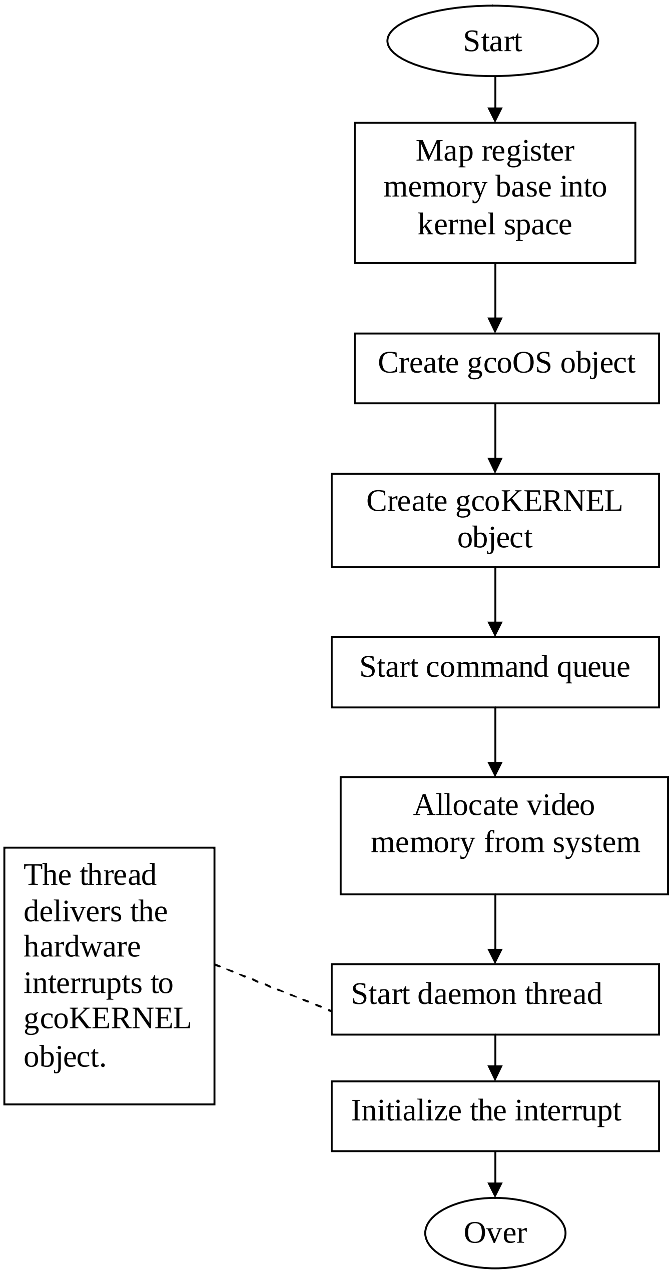
Start
Map register memory base into kernel space
Create gcoOS object
Create gcoKERNEL object
Start command queue
Allocate video memory from system
The thread delivers the hardware interrupts to gcoKERNEL object.
Start daemon thread
Initialize the interrupt
Over

Figure 2 – Kernel driver initialization

1. Register the driver into OS

The driver should provide IOCTL function and this is a communication channel between the user mode and kernel mode. The IOCTL function gets data from the user mode and transfer it to the graphics engine via gcoKERNEL_Dispatch function call. And return the result to the user mode.
Driver is registered into the OS altogether with IOCTL function.

2. Map the register region from IO space into kernel space

Map the register region so that the driver can access the registers in the kernel mode.

3. Create HAL objects

Create gcoOS and gcoKERNEL objects. And start the command queue.

4. Allocate video memory from system

There is no video memory on chip within XAQ2 and the driver pre-allocates video memory from system memory.

5. Interrupt processing

Register the interrupt handler for the irq line. And start a daemon thread to deliver the interrupt to the event manager in the graphics engine. More details please refer to section Interrupt handling.

6. Parameters

User can pass several parameters to the kernel driver, including registerMemBase, irqLine and contiguousSize. registerMemBase is the start address of the register region. irqLine is the irq line number used for the device. And contiguousSize is the video memory size allocated from system memory.

The related code residents in projects/hal/os/linux/kernel/Driver.c. and projects/hal/os/linux/kernel/Device.c. Driver.c contains the registration entry functions and Device.c contains the utility functions.

Interrupt handling

Below is the interrupt processing model.

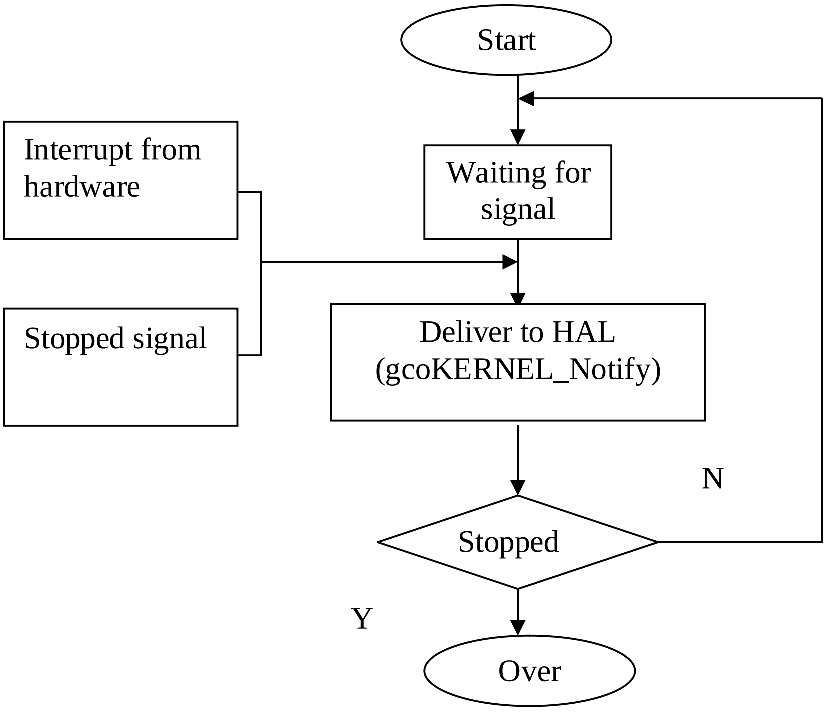

Figure 3 – Interrupt handling model

The daemon thread is created at the initialization stage. It is waiting for signals either from interrupt or termination event. When a hardware interrupt arises, the interrupt handle will be called first. It sets the flag to indicate an interrupt occurred and wakeup the daemon thread. Then the daemon thread delivers the event to HAL layer with gcoKERNEL_Notify function call.
Related code is in projects/hal/os/linux/kernel/Driver.c. and projects/hal/os/linux/kernel/Device.c

Thread management

As thread management is concerned, 3 kinds of API must be implemented.

Thread management: Create thread and destroy thread, etc.
TLS data management: EGL creates the context and stores the pointer into a TLS (Thread Local Storage) variable. And retrieve it as needed.
Synchronization: provide synchronization primitives such as mutex etc. And at the same time we must provide a mechanism to synchronize using the kernel space primitives.

Thread management is contained in EGL module (projects/drivers/openGL/egl/os/eglOSLinux.c) and user OS layer (projects/hal/os/<OS>/user/Os.c).

There are 3 kinds of API on thread management.

1. Thread management

- veglCreateThread

Create a native thread.

- veglColoseThread

Wait for the thread to terminate and then close the handle.

- veglGetCurrentProcessID

Get the current process ID.

2. TLS data management

- veglGetThreadPtr

Get the data pointer stored in TLS. If the data is not created yet, a new one will be created and stored into TLS.

- veglDestroyThread

Remove the data pointer from the TLS.

3. Synchronization

- gcoOS_CreateMutex

Create a new mutex.

- gcoOS_DeleteMutex

Delete a mutex.

- gcoOS_AcquireMutex

Acquire a mutex.

- gcoOS_ReleaseMutex

Release an acquired mutex.

- gcoOS_CreateSignal

Create a new signal which residents in the kernel space.

- gcoOS_DestroySignal

Destroy the signal created with gcoOS_CreateSignal.

- gcoOS_Signal

Set a state of the specified signal.

- gcoOS_WaitSignal

Wait for a signal to become signaled.

Kernel and user OS layers

The OS layer wraps the native OS functions and provides user consistent interfaces.

Kernel mode OS layer has 3 kinds of functions:
1. Memory management
This kind of functions involves memory allocation, memory mapping etc.

- gcoOS_Allocate

Allocate memory from system heap.

- gcoOS_Free

Return the allocated memory back to the system heap.

- gcoOS_AllocateNonPagedMemory

Allocate memory from the non-paged memory zone. This kind of memory is a DMA accessible memory and must not be swapped out.

- gcoOS_FreeNonPagedMemory

Free the memory allocated from the non-paged memory zone.

- gcoOS_AllocatePagedMemory

Allocate memory from the paged pool.

- gcoOS_FreePagedMemroy

Free memory allocated from the paged pool.

- gcoOS_LockPages

Lock memory allocated from the paged pool and return the logical address of the mapped memory.

- gcoOS_UnlockPages

Unlock memory allocated from the paged pool.

- gcoOS_MapPages

Map paged memory into a page table.

- gcoOS_AllocateContiguous

Allocate memory from the contiguous pool.

- gcoOS_FreeContiguous

Free memory allocated from the contiguous pool.

- gcoOS_MapUserPointer

Map a pointer from the user process into the kernel address space.

- gcoOS_UnmapUserPointer

Unmap a user process pointer from the kernel address space.

- gcoOS_WriteMemory

Write data to a memory.

- gcoOS_MapUserMemory

Lock down a user buffer and return an DMA'able address to be used by the hardware to access it.

- gcoOS_UnmapUserMemory

Unlock a user buffer and that was previously locked down by gcoOS_MapUserMemory.

- gcoOS_MapMemory

Map physical memory into the process address space.

- gcoOS_UnmapMemory

Unmap physical memory from the process address space.

- gcoOS_ReadRegister

Read data from a physical register from the underlying hardware.

- gcoOS_WriteRegister

Write data to a physical register from the underlying hardware.

- gcoOS_GetPageSize

Get the page size in the system.

- gcoOS_GetPhysicalAddress

Get the physical address of a corresponding virtual address.

- gcoOS_MapPhysical

Map physical address into kernel space.

- gcoOS_UnmapPhysical

Unmap a previously mapped memory region from kernel memory.

- gcoOS_MemoryBarrier

Make sure the CPU has executed everything up to this point and the data got written to the specified pointer.

2. Synchronization

This kind of functions provides the synchronization primitives within kernel space and supports the user space synchronization utilizing primitives in the kernel space.

- gcoOS_CreateSignal

Create a new signal primitive.

- gcoOS_DestroySignal

Destroy a signal primitive.

- gcoOS_Signal

Set a state of the specified signal. The state can be signaled state or nonsignaled state.

- gcoOS_WaitSignal

Wait for a signal to become signaled.

- gcoOS_MapSignal

Map a signal in to the current process space.

- gcoOS_CreateMutex

Create a mutex primitive.

- gcoOS_DeleteMutex

Delete a mutex primitive.

- gcoOS_AcquireMutex

Acquire a mutex synchronization primitive. The calling thread will be put into sleep until the mutex is available.

- gcoOS_ReleaseMutex

Release an acquired mutex. Make sure any acquired mutex will be released, otherwise a deadlock can occur.

- gcoOS_CreateUserSignal

Create a new signal to be used in the user space.

- gcoOS_DestroyUserSignal

Destroy a signal to be used in the user space.

- gcoOS_WaitUserSignal

Wait for a signal used in the user mode to become signaled.

- gcoOS_SignalUserSignal

Set a state of the specified signal to be used in the user space.

- gcoOS_CleanProcessSignal

Cleanup the proecess’s signal array, which contains the signals used in user space.

3. Misc utility functions

- gcoOS_Construct

Construct a new gcoOS object.

- gcoOS_Destroy

Destroy a gcoOS object.

- gcoOS_Delay

Put the calling thread into sleep for a specified number of microseconds.

- gcoOS_AtomicExchange

Automatically exchange a pair of 32-bit values.

User mode OS layer has 3 kinds of functions:

1. Memory management

- gcoOS_Allocate

Allocate memory from system heap.

- gcoOS_Free

Free the memory allocated with gcoOS_Allocate.

- gcoOS_Reallocate

Reallocate memory from the user heap.

- gcoOS_AllocateNonPagedMemory

Allocate non-paged memory from the kernel

- gcoOS_FreeNonPagedMemory

Free non-paged memory from the kernel.

- gcoOS_MapUserMemory

Lock down a user buffer and return a DMA'able address to be used by the hardware to access it.

- gcoOS_UnmapUserMemory

Unlock a user buffer and that was previously locked down by gcoOS_MapUserMemory.

gcoOS_AllocateNonPagedMemory, gcoOS_FreeNonPagedMemory, gcoOS_MapUserMemory and gcoOS_UnmapUserMemory are the wrappers for the corresponding functioni in the kernel space.

2. Synchronization

- gcoOS_CreateMutex

Create a new mutex.

- gcoOS_DeleteMutex

Delete a mutex.

- gcoOS_AcquireMutex

Acquire a mutex.

- gcoOS_ReleaseMutex

Release an acquired mutex.

- gcoOS_CreateSignal

Create a new signal which residents in the kernel space.

- gcoOS_DestroySignal

Destroy the signal created with gcoOS_CreateSignal.

- gcoOS_Signal

Set a state of the specified signal.

- gcoOS_WaitSignal

Wait for a signal to become signaled.

gcoOS_CreateSignal, gcoOS_DestroySignal, gcoOS_Signal and gcoOS_WaitSignal are the wrappers for the corresponding functioni in the kernel space.

3. Misc utility functions

- gcoOS_Construct

Construct a new gcoOS object.

- gcoOS_Destroy

Destroy a gcoOS object.

- gcoOS_DeviceControl

Perform a device I/O control call to the kernel API.

- gcoOS_LoadLibrary

Load a library dynamically.

- gcoOS_FreeLibrary

Unload a dynamically loaded library.

- gcoOS_GetProcAddress

Get the address of a function inside a loaded library.