SGX DDK for Android

Software Functional Specification

Filename	:	SGX DDK for Android.Software Functional Specification (1.8 DDK).docx
Version	:	1.8.18 External Issue
Issue Date	:	26 Jan 2012
Author	:	Imagination Technologies

Contents

List of Figures

1. Introduction

1.1. Scope

Since most of the software components within the SGX DDK have separate detailed Functional Specifications this document is largely an overview of the constituent components within the DDK with references to the more detailed documents.

The Android platform differs significantly from other Linux-based systems. The Android user space is neither source nor binary compatible with GNU/Linux systems and presents unique porting issues. The Android graphics architecture is likewise unique to Android and bears no resemblance to other existing Linux GUI architectures

1.2. Related Documents

SGX Embedded Systems DDK for Linux, Software Functional Specification
OpenGL-ES Specification, Khronos Group, Version 1.0, 1.1, 2.0
EGL Specification, Khronos Group, Version 1.4
SGX OpenGL ES 2.0 Reference Driver, Software Functional Specification
SGX OpenGL ES 1.1 Reference Driver, Software Functional Specification
SGX EGL 1.x Reference Driver, Software Functional Specification
Pluggable Window System.Implementation Guide
Pluggable Window System.API

1.2.1. Linux SFS Applicability

The SGX Embedded Systems DDK for Linux, Software Functional Specification document applies largely to Android, in particular the detailed coverage of the SGX DDK Build system. A section in this document outlines options which are unique to Android.

2. Component Structure

The following diagram details a top-level overview of the major components of the SGX DDK for Android and their relationships.

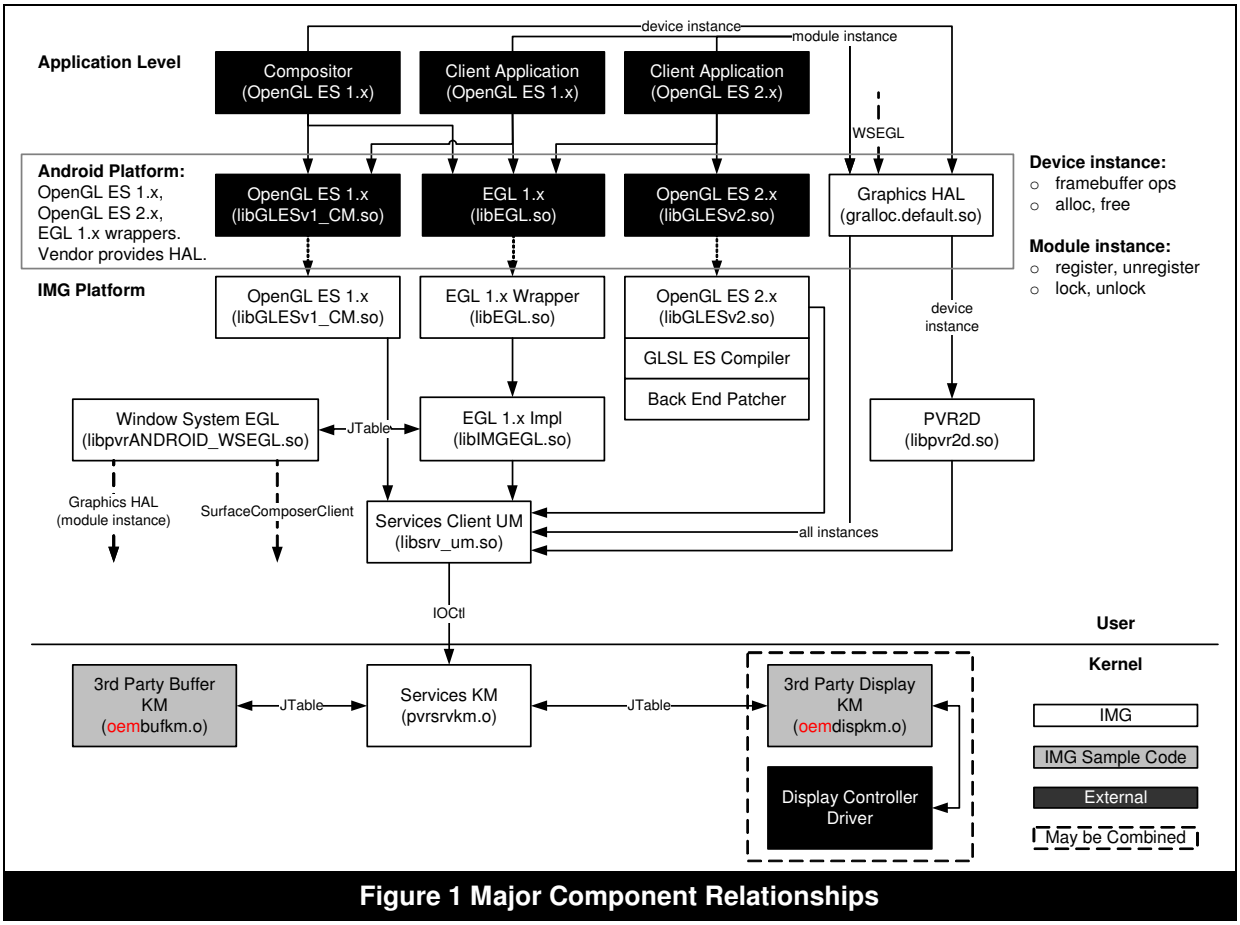


Figure 1 Major Component Relationships

3. Generic Platform Overview

Drivers integrating with the Android platform must implement the components listed below.
The reference software renderer and the SGX driver are both implemented in this way.

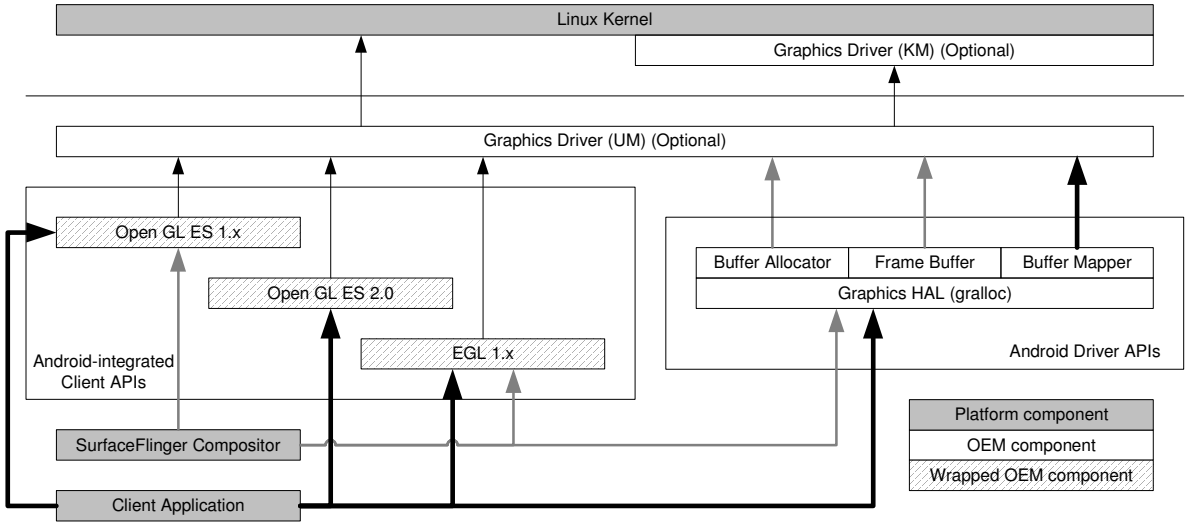


Figure 2 Android Graphics Driver Model

3.1. Graphics HAL (gralloc)

This graphics HAL component is an abstraction layer. It has the following responsibilities:

- Buffer management
  - Allocation and destruction of buffers
  - Mapping and un-mapping of buffers
  - Constraining buffer pixel format and other attributes
- Frame buffer I/O
  - Mode setting (or mode detection/inheritance if not supported)
  - Allocation and destruction of frame buffers
  - Presentation blitting or flipping, buffer posting
- CPU/GPU synchronization (optional for SW renderer)
  - Locking buffers (mutual exclusion)
  - Cache flushing
- Composition completion hook (optional for any driver)
  - Called when the compositor finishes a frame, before `eglSwapBuffers()`.

When a buffer is created, the graphics HAL must assign it a unique handle that can be communicated cross process (Android's `native_handle_t` type). This creates an incompatibility between graphics HAL implementations. As such, there is typically only one graphics HAL in use on the system at any time.

3.2. Client APIs

Android natively supports Open GL ES 1.1, Open GL ES 2.0 and EGL 1.4.

For each of these client APIs, the vendor must provide an implementation which is loaded at runtime. Applications link to the platform libraries, and calls to client API functions are directed through a vendor implementation responsible for a context, derived from an application-selected `EGLConfig`.

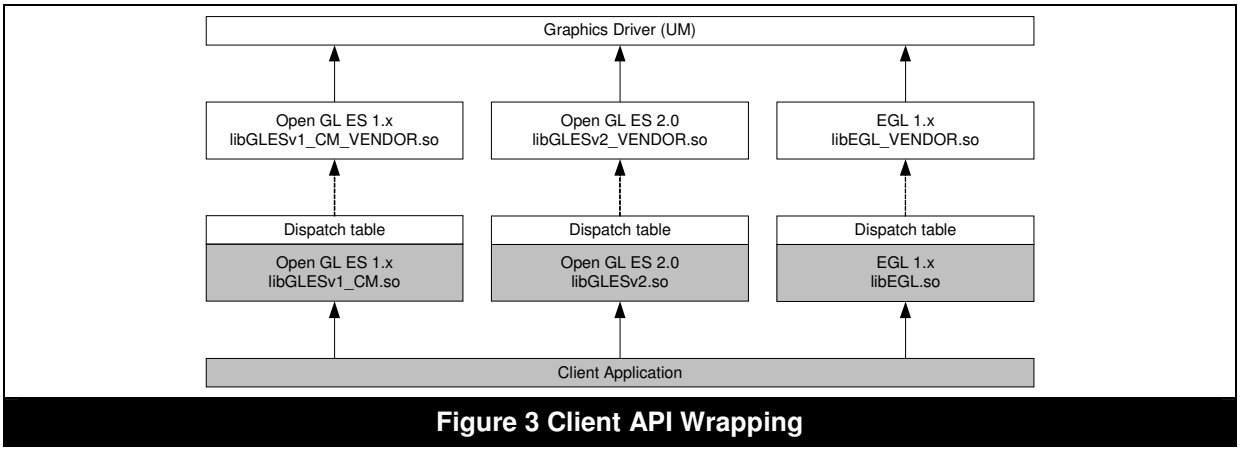


Figure 3 Client API Wrapping

For this reason, there may be multiple implementations of these client APIs on the system. A common configuration is the reference software EGL/ES coupled with a hardware accelerated EGL/ES implementation. With such combinations, an aggregated list of `EGLConfig` is returned to the application.

3.3. Important Platform Data Types

3.3.1. EGL Native Types

- `EGLNativeDisplayType` is mapped to `void*`
  Android has no formal concept of a native display.
- `EGLNativeWindowType` is mapped to `android_native_window_t`, a core UI type.
- `EGLNativePixmapType` is mapped to `egl_native_pixmap_t` which is a legacy feature and new applications do not use it.

3.3.2. Other Graphics Types

- `android_native_window_t` is used to express uniqueness/multiplicity of windows.
  It has methods such as `dequeueBuffer()`, `queueBuffer()`, `lockBuffer()` which have different meanings depending on whether the window is a client window or a frame buffer window (typically used only by the compositor).
- `android_native_buffer_t` is a type used to obtain a render surface for a given frame.
  Each buffer has metadata such as width, height, stride, format and usage bits, and a *handle* which can be used by the driver, to map and unmap pixel data.

3.4. Cross-Process Communication

The Android compositor (SurfaceFlinger) is solely responsible for all buffer allocations. These allocations are specified by client applications, over IPC, prior to EGL initialization.

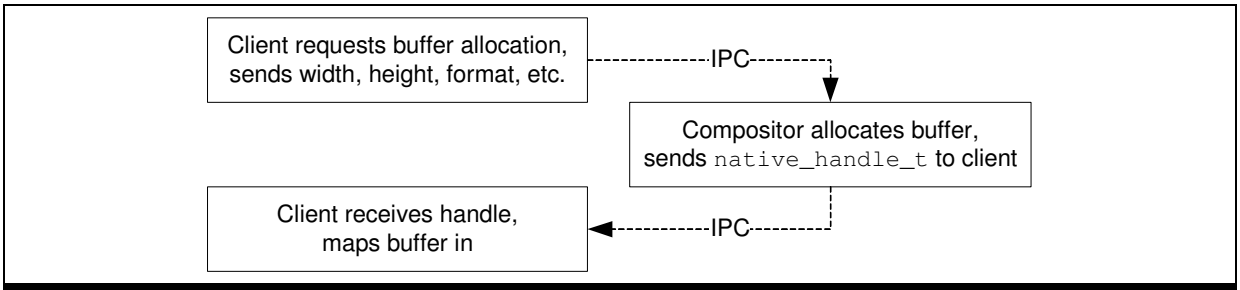


Figure 4 Typical SurfaceFlinger Buffer Allocation Protocol

In order to keep the system manageable and secure, each buffer is assigned a handle which should globally (uniquely) identify it in the system. This handle is returned to the client application via the same IPC.

The client and compositor map the allocation into their address space using these handles. The format and delivery of these handles is handled by the Android platform, but the mapping is done by the vendor's graphics HAL module in both processes.

For more information about these handles, see the Android `native_handle_t` type.

4. Additional SGX DDK Components

The SGX driver includes some additional components for the Android platform that are not present on Linux.

4.1. ANDROID_WSEGL

This is a custom pluggable Window System EGL module for the Android platform. It is used by both client EGL applications and the compositor.

This module interacts with Android's `libui` to queue/dequeue buffers, validate pixel formats and map them to SGX HW formats, and buffer posting (double-buffered windows). It is also responsible for window resizing and reference counting.

4.2. libtestwrap

This module is not strictly part of the DDK. Intended for the DDK unit test infrastructure, it allows native EGL and Open GL ES applications to be more easily ported to Android. It uses the Android platform types and APIs to provide a simple, consistent C API for writing test cases and porting larger applications.

The SGX DDK for Android ships with tests which can be composited by SurfaceFlinger or rendered directly to the frame buffer, if no compositor is running. The `testwrap` binary makes this seamless and automatic.

5. SGX DDK Features

The SGX driver implements some additional features that apply only to Android and not mandatory for the platform. They enhance security and performance, and tighten platform integration.

5.1. Cross-Process Security

To keep buffers secure, the SGX driver backs buffer allocations with UNIX file descriptors which are duplicated in the receiving process by Android's IPC. These handles are valid only in the process that requested the buffer and the compositor (which allocated it). No other client process can map this buffer.

The SGX driver creates secure, per-process GPU mappings for each allocated buffer. For any rendering process, the SGX MMU is programmed from per-process page tables.

These features prevent applications from faking handles and/or mapping buffers that they do not own, in order to read/write to them.

The Android platform does not enforce this kind of security; other implementations may be exploitable.

5.2. Zero-Copy Composition

Android supports two composition methods. The first, a legacy method used in Donut and earlier versions uses `glTexImage2D()` each frame to load client application buffers. This involves a potentially expensive (CPU) copy from the client buffer into the compositor's GL context. This method also increases serialization, which is costly for deferred rendering architectures, even if it does simplify synchronization.

Introduced in Éclair is the Android-only `EGL_ANDROID_image_native_buffer` extension which allows `eglCreateImageKHR()` to take an `android_native_buffer_t` "target" parameter. Coupled with the other base `EGL_image` extensions, this allows an Android buffer to make it into the compositor's GL without any copies, and puts more synchronization in the hands of the driver.

The SGX driver implements some complex synchronization and cache flushing logic to optimize composition for deferred rendering. This increases performance and reduces the dependency on CPU power.

5.3. 'Make' System Integration

The Android SGX driver tightly integrates with the Android GNU make system. The driver is built with the Android tool chain (compiler, assembler, linker, etc.) and against the Android C library (bionic). The Android platform headers are used for all modules. This ensures better binary compatibility with the platform and platform debugging facilities.

6. Android-specific Application Hints

As with other platforms, application hints (AppHints) can be written into /system/etc/powervr.ini to influence driver behaviour on a per-process basis.

See the Linux SFS for more information about generic hints.

Android adds three additional hints which are primarily used for debugging and should not be necessary for production devices:

- **HALPresentMode=N**
  Where N is 0 (default), 1 or 2. 0 is Flipping. 1 is Blitting. 2 is front-buffer rendering.
- **HALNumFrameBuffers=N**
  This affects the length of the flip chain or number of back-buffers for blitting. The default is 2. If set to <2 this will force-enable front-buffer rendering.
  NOTE: Increasing the number of allocated buffers does not guarantee the framework will use them. The framework must be modified to use the additional buffers.
- **HALFrameBufferHz=N**
  This is cosmetic and represents the value stored in the window->fps field. The default is 68.

- **HALCompositionBypass=N**
  Where N is 0 (off) or 1 (on, default). When set to 1, the graphics HAL will honour allocation requests of HW_FB | HW_RENDER and can support PBE-transformed renders to these buffers in the Android Window System EGL module.
  See the **Full Screen Compositor Bypass** document for more information on bypass.

7. Android-specific PDUMP Information

7.1. Hardware PDUMPs

To generate a hardware PDUMP of an application under Android, create or modify `/system/etc/powervr.ini` in the target file-system image to specify the target application(s). For example:

```
[my_process_name]
PDumpActive=1
```

Start the PDUMP utility from another console (either the serial console or `adb shell`):

```
pdump -c100
```

Finally, start the application from the interface as normal.

7.2. No-hardware PDUMPs

No-hardware in this context refers to “no SGX”. Use of Android’s SurfaceFlinger is not supported; only direct-to-framebuffer use cases can work (via testwrap or similar).

1. Build the DDK with the `NO_HARDWARE=1 PDUMP=1 SUPPORT_PDUMP_MULTI_PROCESS=0` options and install it
2. Disable processes spawned by `init` that generate rendering.
   Edit `$ANDROID_ROOT/out/target/product/blaze/root/init.rc` and comment out the `zygote` and `surfaceflinger` services.
3. Rebuild the `boot.img` file so that the ramdisk is updated with the new `init.rc`:

```
$ make -j4 TARGET_PRODUCT=blaze TARGET_BUILD_TYPE=release TARGET_BUILD_VARIANT=userdebug \
       out/target/product/blaze/boot.img
```

4. Re-flash (to the Blaze) the `boot.img` with fastboot:

```
$ fastboot flash boot out/target/product/blaze/boot.img
```

5. Follow the instructions in section 7.1.

Note that no output will be displayed on the screen with a no-hardware driver.

8. Listing of Makefile Variables

These variables augment the variables in the base Linux SFS documentation.

8.1. Variables affecting the toolchain flags

Name	Set in	Default	Description
TARGET_PRODUCT	Makefile	generic	Tells make system where to look for libraries, etc. Must match option passed to Android platform build
TARGET_BUILD_TYPE	paths.mk	debug	Tells make system where to look for libraries, etc. Must match option passed to Android platform build Note: This does not affect the SGX DDK build type

8.2. Variables affecting names and destinations of generated files

Name	Set in	Default	Description
GRALLOC_VARIANT	Makefile extra_config.mk	default	Forms the name of the graphics HAL module i.e. gralloc.$(GRALLOC_VARIANT).so
OGLES1_V1_ BASENAME	extra_config.mk	GLESv1_CM_POWERVR_ SGX$(SGXCORE)_ $(SGXCOREREV)	Forms the name of the wrapped GLESv1 library
OGLES2_BASENAME	extra_config.mk	GLESv2_POWERVR_ SGX$(SGXCORE)_ $(SGXCOREREV)	Forms the name of the wrapped GLESv2 library
EGL_BASENAME	extra_config.mk	EGL_POWERVR_ SGX$(SGXCORE)_ $(SGXCOREREV)	Forms the name of the wrapped EGL library

9. IMG Graphics HAL

This section details the structure and implementation of the IMG graphics HAL module for Android.

9.1. Structural Overview

Although the graphics HAL is a single dynamically shared object, it comprises of multiple functional components that provide different APIs to the callers:

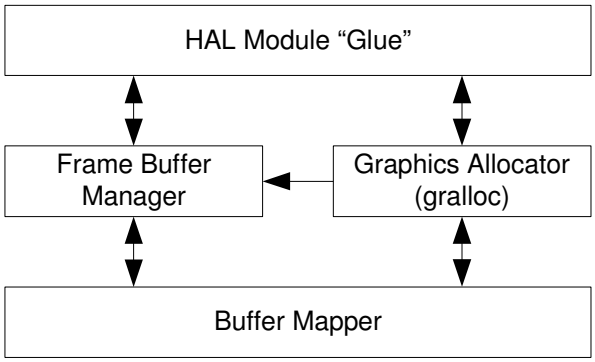


Figure 5 Component modules of IMG graphics HAL

9.1.1. HAL Module “Glue”

Relevant files: hal.c, hal.h, hal_private.h

This component contains the HAL_MODULE_INFO_SYM which is a symbol that the Android framework will try to find in any HAL module.

The symbol corresponds to a structure which contains the Android GRALLOC module identifier, various mandatory API functions, and some IMG API extensions.

The component is also responsible for maintaining a connection to PVR Services. A connection is established for any application that loads the HAL (compositor or client) and is closed if the module is unloaded.

9.1.2. Buffer Mapper

Relevant files: mapper.c, mapper.h

This component contains a table of all graphics buffers created or mapped by the process. Each entry in the table is a buffer, and each buffer has metadata associated with it.

This metadata comprises of a key (the buffer "stamp"), the usage bits the surface was allocated with (e.g. SW/HW rendering), a reference lock count, lock usage bits (subset of usage bits) and a write-lock rectangle for optimized cache flushing.

9.1.3. Frame Buffer Manager

Relevant files: framebuffer.c, framebuffer.h

This component manages frame buffer allocation and destruction. At present, this component will pre-allocate a flip chain (if possible) on open, and pass along the buffers to gralloc when alloc() is called.

If flipping isn't available, the primary framebuffer will be returned for all alloc() requests.

It implements the setSwapInterval(), post() and compositionComplete() gralloc API functions.

Any buffers allocated by this component are added to the buffer mapper.

It is only used by the compositor, not client applications.

9.1.4. Graphics Allocator (gralloc)

Relevant files: gralloc.c, gralloc.h

This component manages the allocation and destruction of all other buffers. Any buffers allocated by this component are added to the buffer mapper.

It implements the alloc() and free() gralloc API functions for use by the compositor.

It implements the lock(), unlock(), registerBuffer(), unregisterBuffer() gralloc API functions for use by client applications.

This component is responsible for maintaining proper GPU<->CPU cache coherency.

9.2. IMG-specific Extensions

The IMG graphics HAL implementation extends some aspects of the original graphics HAL spec as required by our EGL implementation.

The gralloc API extends the base hw_module_t structure to add lock(), unlock(), unregisterBuffer() and registerBuffer() entry points. This new structure is called gralloc_module_t. This is the bare minimum a graphics vendor must implement.

The IMG graphics HAL further extends this with map(), unmap() and various debugging functions. This new structure is called IMG_gralloc_module_t and encapsulates gralloc_module_t. These extension functions are used by the IMG EGL implementation and are implemented in the Graphics Allocator component.

Additionally, some private data is stored in the IMG_gralloc_module_t. This private data contains various fields for interacting with PVR services, a mutex to implement graphics HAL re-entrancy, and the base pointer for the Buffer Mapper.

9.3. Implementation of native_handle_t

To allow graphics vendors some implementation flexibility, the Android platform defines a data type which it will send/receive over Binder IPC. The IMG implementation of this type is known as **IMG_native_handle_t** and its definition is shown below:

```
typedef struct
{
        native_handle_t base;
        int fd;
        IMG_UINT64 ui64Stamp;

        int usage;
        int width;
        int height;
        int bpp;
}
IMG_native_handle_t;
```

Instances of these handles are “created” by the graphics HAL (when a buffer is allocated) and can be interpreted either by the graphics HAL or the vendor’s EGL implementation.

This data type is proprietary to the IMG graphics HAL implementation and interoperability with other vendor’s modules is neither required nor guaranteed.

9.3.1. Field “base”

All implementations of native_handle_t must have this field listed first. This field contains some private data which the Android framework uses to reference count users of the handle. The handle may be destroyed by the framework when this reference count falls to zero.

9.3.2. Field “fd”

This is a secure file descriptor that can be used to map a graphics buffer into the calling process. This is not a globally unique numeric, because file-descriptors are per-process. All buffers except frame buffers have a valid file descriptor; frame buffers will have a value of -1 here.

9.3.3. Field “ui64Stamp”

This is a globally unique ID for the graphics buffer. It is used as a key to the Buffer Mapper (primarily as an optimization) and is not used directly to map a buffer. All buffers have a stamp.

9.3.4. Other Fields

The other fields in this handle are essentially metadata, used to simplify client-side cache coherency code. These fields are not particularly notable and may be removed as future versions of Android provide the necessary information.

10. ANDROID_WSEGL

This section details the implementation of the IMG Window System EGL module for Android.

10.1. Users

The WSEGL module is used by the compositor for rendering to the frame buffers directly, and the same module is used by client applications for rendering to off-screen buffers.

In most cases, the module’s interface to the platform is used identically for compositor and client. However, there are some important exceptions, detailed below.

10.2. Interaction with Graphics HAL

The ANDROID_WSEGL module interacts with the HAL both directly and indirectly. It is implied that any Android client application using EGL will always have the graphics HAL loaded.

The HAL is used directly in GetDrawableParameters() in order to map() and unmap() the render target buffers. This allows user-virtual and device-virtual addresses to be returned to EGL.

The HAL is used indirectly to inspect the “fd” field from any native_handle_t. This is used to differentiate frame buffers from all other buffers.

10.3. Supported Drawable Types

An EGLSurface can be created from an android_native_window_t with eglCreateWindowSurface().

10.4. Supported Pixel Formats

ANDROID_WSEGL advertises all pixel formats that are mutually compatible with the Android framework and SGX hardware:

Android Pixel Format	Supported by DDK?
RGBA 8888	Yes
RGBX 8888	Yes
RGB 888	No
RGB 565	Yes
BGRA 8888	Yes
RGBA 5551	No (BGRA 5551 only)
RGBA 4444	No (BGRA 4444 only)

This list has an effect on the available EGLConfigs.

Any call to eglCreateWindowSurface() for an android_native_window_t with a format different to either the selected EGLConfig or the supported formats in the above table will fail with EGL_BAD_MATCH.

11. Known Issues

11.1.1. Defunct setSwapInterval() for Client Applications

The Android platform is responsible for wiring up the setSwapInterval() function on android_native_window_t's it allocates.

For frame buffer windows (i.e. the compositor's render targets) the function is wired up to call setSwapInterval() in the IMG Frame Buffer Manager component.

However, for off-screen render targets used by client applications, there is no specified setSwapInterval() behaviour, and the usual default of 1 is not guaranteed.

11.1.2. Unusable EGLConfigs Returned

Android provides no clearly reliable distinction between compositor and client processes. As a result, there is no way to know whether an EGL context will be used for composition or client rendering prior to eglCreateWindowSurface(). Therefore, it is not possible to constrain the list of EGLConfig returned by eglGetConfig()/eglChooseConfig().

By default, the entire list of configs available to *client* applications will be returned, even to the compositor. It is the responsibility of the compositor to ensure that it selects an EGLConfig which matches the format of the frame buffer. The Android platform provides a helper function to achieve this.

11.1.3. Compositor's eglSwapBuffers() Blocks

For client application 3D renders, many EGL/GLES functions will not predictably block. They will only block if a hardware resource is exhausted or there is an outstanding render dependency. This behaviour is deliberate, in order to maximise queuing and improve the efficiency of deferred rendering.

The relative unpredictability of these blocks can have an adverse effect on the Android composition architecture. If a GLES function blocks inside the composition "critical section" (when all client windows are locked) this prevents client applications from de-queuing buffers and initiating renders.

For renders that can occur asynchronous to HW composition (such as SW rendering), blocking in this critical section introduces a significant performance penalty. Fortunately, the call to eglSwapBuffers() lies outside of this critical section, when the client windows are unlocked.

ANDROID_WSEGL will detect if eglSwapBuffers() has been called with a frame buffer window current, and cause that render to be flushed before unblocking. This makes the "blocks" occur predictably in eglSwapBuffers() for typical composition renders.

One negative side-effect of this change is that compositor TAs can no longer overlap with compositor 3Ds. However, as compositor TAs are very short (~1ms typical) in practice this is a minor issue.

11.1.4. Clients “Locked” to Compositor Frame Rate

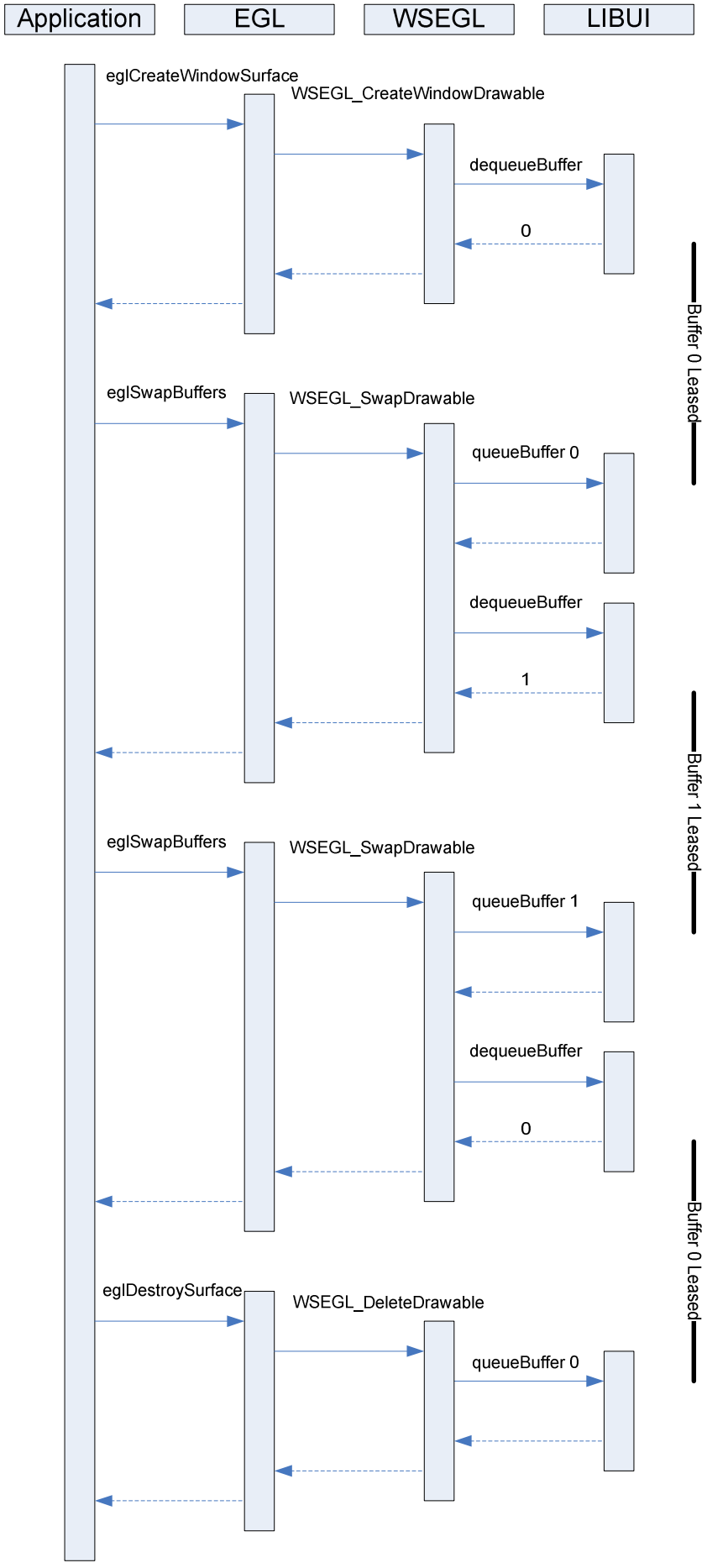
The Android platform requires that graphics vendors obtain render buffers with dequeueBuffer() and return them to the pool with queueBuffer(). Calling queueBuffer() also marks the window contents as updated and flags the buffer for composition.

This mechanism is designed to be friendly to deferred renderers as it allows buffers to be submitted for composition asynchronously.

For various technical reasons, and to enhance general-case performance, the SGX driver performs some additional render buffer synchronization (henceforth referred to as “sync objects”). Sync objects supersede the platform’s synchronization mechanism.

In order to improve sync object efficiency, it is necessary to return buffers to the compositor before the client render has technically finished. Immediately after the client render is started, the target buffer is returned to the compositor with queueBuffer(). If the compositor is idle, it will immediately start a composition involving the unfinished render. As the client application has not yet finished (or perhaps even started) its render, the compositor will be forced to wait until the client render completes, before immediately starting the composition render.

Sync objects ensure that all renders complete in the correct sequence. When the compositor render is queued, the sync object mechanism prevents another render starting on the surface until the compositor has finished reading it. This effectively means the client frame-rate cannot exceed compositor frame-rate in most applications, because after it has exhausted both buffers, it can no longer dequeue buffers. This has an adverse effect on some benchmarks, but improves the performance of non-benchmark applications.


Application
EGL
WSEGL
LIBUI
eglCreateWindowSurface
WSEGL_CreateWindowDrawable
dequeueBuffer
0
Buffer 0 Leased
eglSwapBuffers
WSEGL_SwapDrawable
queueBuffer 0
dequeueBuffer
1
Buffer 1 Leased
eglSwapBuffers
WSEGL_SwapDrawable
queueBuffer 1
dequeueBuffer
0
Buffer 0 Leased
eglDestroySurface
WSEGL_DeleteDrawable
queueBuffer 0