Khronos API Implementers Guide

Fourth Edition

Edited by Mark Callow
HI Corporation

Abstract

Records guidelines to be followed by implementers of Khronos Group *APIs.*

Table of Contents

1. Introduction

This document provides guidelines for *implementers* of OpenGL ES, OpenVG and other *API* standards specified by the Khronos Group. The aim of these hints is to provide commonality between implementations to ease the logistical problems faced by developers using multiple different implementations of an *API*.

One of the primary goals is to allow an application binary to run on top of multiple different implementations of an *API* on the same platform.

Implementers are strongly urged to comply with these guidelines.

This document contains links to several Khronos specifications and header files at versions current at the time of publication. Readers may wish to check the Khronos registry[1] for post-publication updates to these files.

2. Header Files and Libraries

2.1. Uncontrolled Platforms (e.g. GNU®/Linux®, Windows®, Windows CE)

When providing implementations for platforms where the *platform vendor* does not provide or has not yet established a standard *Application Binary Interface* for an *API*, *implementers* are strongly urged to comply with the following recommendations. *Vendors* newly establishing *ABI* specifications for their platforms are also strongly urged to comply with these recommendations.

2.1.1. Header Files

2.1.1.1. General

- *Implementers* are strongly encouraged to use the standard header files (`egl.h`, `gl.h`, `kd.h`, etc.) for each specification that are provided by Khronos and listed in Table 1, “Header File Names and Locations”. Links are provided there.
- Portable and non-portable definitions are separated into <api>.h and <api>platform.h files, e.g., `gl.h` and `glplatform.h`. Implementers should not need to change the former; they are strongly discouraged from doing so. They should rarely need to change the latter as it already contains definitions for most common platforms.
- Khronos provides a common underlying `khrplatform.h` defining sets of base types, linkage specifications and function-call specifications covering most common platforms. Many Khronos-provided <api>platform.h and other header files include `khrplatform.h` and use its definitions as the basis for their own. *Implementers* should rarely need to change this file.
- For most *APIs*, functions and enumerants for extensions registered with the Khronos Extension Registry are declared and defined in a Khronos-provided <api>ext.h file, e.g, `glext.h`. Exceptions are noted below. These header files can be used even if the implementation doesn't provide a particular extension since applications must query the presence of extensions at run time.
- Functions and enumerants for unregistered *implementer* extensions should be declared and defined in an *implementer's* own header file. Follow the extension writing and naming rules given in *How to Create Khronos API Extensions*[2]. Use enumerant values obtained from the Khronos Extension Registry, as explained in *OpenGL Enumerant Allocation Policies*[3]. Implementers are strongly encouraged to register their extensions even if they are the only vendor.
- Some *APIs* define optional utility libraries. Functions and enumerants for these are declared and defined in Khronos-provided header files, e.g., `glu.h`. For some other *APIs*, header file names are reserved for use by future utility libraries.

[1] http://www.khronos.org/registry/
[2] http://www.opengl.org/registry/doc/rules.html
[3] http://www.opengl.org/registry/doc/enums.html

- Header files for a given *API* are packaged in an API-specific folder, e.g., `GLES`. Folder names are listed in Table 1, "Header File Names and Locations" All API specific folders should be placed in a common parent folder.
- Consider contributing any header file changes back to Khronos so that others may benefit from your expertise. Contact the relevant working group and, on approval, update the files in the Khronos Subversion tree.
- Use the function-call convention defined for the platform in the Khronos-provided <api>platform.h, if a definition exists for the platform in question.
- If you make your own header files use the names given in Table 1, "Header File Names and Locations".
- If the platform is Windows or WindowsCE, make sure your header files are suitable for use with *MFC*. Make sure that any base API types referred to are preceded by `::`. For example you need to refer to `::GetDC(0)` because several Microsoft Foundation Classes have their own `GetDC(void)` methods.
- When including one header in another, include the parent directory name. For example when including `eglplatform.h` in `egl.h` use `#include <EGL/eglplatform.h>`. Do not use `#include <eglplatform.h>` because it forces application makefiles to specify 2 different `-I<path>` options to find both include files.

Table 1. Header File Names and Locations

API	Location	Header Files	How to include	Provider
OpenCL	`CL`	`cl.h`[4]	`#include <CL/cl.h>`	Khronos
		`cl_gl.h`[5a]	`#include <CL/cl_gl.h>`	Khronos
		`cl_platform.h`[6]	Included by `cl.h`	Khronos, possibly modified by *Vendor* or *Implementer*
		`cl_ext.h`	`#include <CL/cl_ext.h>`	Khronos, once extensions exist
EGL 1.x	`EGL`	`egl.h`[7]	`#include <EGL/egl.h>`	Khronos
		`eglplatform.h`[8 b]	Included by `egl.h`	Khronos, possibly modified by *Vendor* or *Implementer*
		`eglext.h`[9]	`#include <EGL/eglext.h>`	Khronos
OpenGL® 3.x & 4.x Compatibility	`GL`	`gl.h`	`#include <GL/gl.h>`	Platform or Implementer
		`glext.h`[10]	`#include <GL/glext.h>`	Khronos
		`glu.h`[c]	`#include <GL/glu.h>`	Implementer

[4] http://www.khronos.org/registry/cl/api/1.0/cl.h
[5] http://www.khronos.org/registry/cl/api/1.0/cl_gl.h
[6] http://www.khronos.org/registry/cl/api/1.0/cl_platform.h
[7] http://www.khronos.org/registry/egl/api/egl.h
[8] http://www.khronos.org/registry/egl/api/eglplatform.h
[9] http://www.khronos.org/registry/egl/api/eglext.h
[10] http://www.opengl.org/registry/api/glext.h

API	Location	Header Files	How to include	Provider
OpenGL® 3.1+ Core	GL	glcorearb.h[11d]	#include <GL/glcore-arb.h>	Khronos
		glcoreext.h[e]	#include <GL/glcoreext.h>	Khronos / Implementer
		glu.h[c]	#include <GL/glu.h>	Implementer
OpenGL® ES 1.x	GLES	gl.h[12]	#include <GLES/gl.h>	Khronos
		glplatform.h[13 f]	Included by gl.h	Khronos, possibly modified by *Vendor* or *Implementer*
		glext.h[14]	#include <GLES/glext.h>	Khronos
		glu.h	Reserved for future use	
OpenGL® ES 2.x	GLES2	gl2.h[15]	#include <GLES2/gl2.h>	Khronos
		gl2platform.h[16]	Included by gl2.h	Khronos, possibly modified by *Vendor* or *Implementer*
		gl2ext.h[17]	#include <GLES2/gl2ext.h>	Khronos
		glu{,2}.h	Reserved for future use	
OpenGL® ES 3.x	GLES3	gl3.h[18]	#include <GLES3/gl3.h>	Khronos
		gl3platform.h[19]	Included by gl3.h	Khronos, possibly modified by *Vendor* or *Implementer*
		gl3ext.h[20]	#include <GLES3/gl3ext.h>	Khronos
		glu{,3}.h	Reserved for future use	
*	KHR	khrplatform.h[21]	Included by header files which use it. Never included directly by applications.	Khronos

[11] http://www.opengl.org/registry/api/glcorearb.h
[12] http://www.khronos.org/registry/gles/api/1.1/gl.h
[13] http://www.khronos.org/registry/gles/api/1.1/glplatform.h
[14] http://www.khronos.org/registry/gles/api/1.1/glext.h
[15] http://www.khronos.org/registry/gles/api/2.0/gl2.h
[16] http://www.khronos.org/registry/gles/api/2.0/gl2platform.h
[17] http://www.khronos.org/registry/gles/api/2.0/gl2ext.h
[18] http://www.khronos.org/registry/gles/api/3.0/gl3.h
[19] http://www.khronos.org/registry/gles/api/3.0/gl3platform.h
[20] http://www.khronos.org/registry/gles/api/3.0/gl3ext.h
[21] http://www.khronos.org/registry/egl/api/khrplatform.h

API	Location	Header Files	How to include	Provider
OpenKODE 1.x	KD	kd.h[22]	#include <KD/kd.h>	Khronos
		kdplatform.h[23]	Included by kd.h	Khronos, possibly modified by *Vendor* or *Implementer*
OpenMAX AL	OMXAL	OpenMAXAL.h[24]	#include <OMXAL/OpenMAXAL.h>	Khronos
		OpenMAXAL_Platform.h[25]	Included by OpenMAXAL.h	Khronos, possibly modified by *Vendor* or *Implementer*
		OpenMAXAL_IID.c[26g]	Compile and link with the application	Khronos
OpenSL ES	SLES	OpenSLES.h[27]	#include <SLES/OpenSLES.h>	Khronos
		OpenSLES_Platform.h[28]	Included by OpenSLES.h	Khronos, possibly modified by *Vendor* or *Implementer*
		OpenSLES_IID.c[29g]	Compile and link with the application	Khronos
OpenVG 1.x	VG	openvg.h[30]	#include <VG/openvg.h>	Khronos
		vgplatform.h[31]	Included by openvg.h	Khronos
		vgext.h[32]	#include <VG/vgext.h>	Khronos
		vgu.h[33 h]	#include <VG/vgu.h>	Khronos
OpenWF	WF	wfc.h[34]	#include <WF/wfc.h>	Khronos
		wfcplatform.h[35]	Included by wfc.h	Khronos, possibly modified by *Vendor* or *Implementer*
		wfcext.h	#include <WF/wfcext.h>	Khronos

[22] http://www.khronos.org/registry/kode/api/kd.h
[23] http://www.khronos.org/registry/kode/api/kdplatform.h
[24] http://www.khronos.org/registry/omxal/api/1.1/OpenMAXAL.h
[25] http://www.khronos.org/registry/omxal/api/1.1/OpenMAXAL_Platform.h
[26] http://www.khronos.org/registry/omxal/api/1.1/OpenMAXAL_IID.c
[27] http://www.khronos.org/registry/sles/api/1.1/OpenSLES.h
[28] http://www.khronos.org/registry/sles/api/1.1/OpenSLES_Platform.h
[29] http://www.khronos.org/registry/sles/api/1.1/OpenSLES_IID.c
[30] http://www.khronos.org/registry/vg/api/1.1/openvg.h
[31] http://www.khronos.org/registry/vg/api/1.1/vgplatform.h
[32] http://www.khronos.org/registry/vg/api/1.1/vgext.h
[33] http://www.khronos.org/registry/vg/api/1.1/vgu.h
[34] http://www.khronos.org/registry/wf/api/1.0/openwfc.1.0.headers.zip
[35] http://www.khronos.org/registry/wf/api/1.0/openwfc.1.0.headers.zip

API	Location	Header Files	How to include	Provider
		wfd.h[36]	#include <WF/wfd.h>	Khronos
		wfdplatform.h[37]	included by wfd.h	Khronos, possibly modified by *Vendor* or *Implementer*
		wfdext.h	#include <WF/wfdext.h>	Khronos

[a]A file declaring the functions used to integrate OpenCL and OpenGL objects.

[b]Early EGL implementations used egltypes.h instead of the now recommended eglplatform.h .

[c]Required, if the OpenGL utility library is provided.

[d]Includes only interfaces for the Core profile of OpenGL and ARB extensions.

[e]Does not currently exist as there are no known vendor extensions to the Core profile.

[f]glplatform.h does not exist in many early implementations of OpenGL ES 1.x. Platform dependent declarations were included directly in gl.h .

[g]This file contains unique interface IDs for all API interfaces. These IDs have been automatically generated. Neither implementers nor application developers should edit these interface IDs.

[h]Required, if the OpenVG utility library is provided.

2.1.1.2. Notes

2.1.1.2.1. General

- To find the include files, use appropriate compiler options in the makefiles for your sample programs; e.g. *-I* (gcc, linux) or */I* (Visual C++).
- Given the different *IDEs* & compilers people use, especially on Windows, it is not possible to recommend a system location in which to place these include files. Where obvious choices exist Khronos recommends implementers take advantage of them.
- In particular, GNU/Linux implementations should follow the *Filesystem Hierarchy Standard* [38] for the location of headers and libraries.

2.1.1.2.2. EGL

- Implementers may need to modify eglplatform.h. In particular the eglNativeDisplayType, eglNativeWindowType, and eglNativePixmapType typedefs must be defined as appropriate for the platform (typically, they will be typedef'ed to corresponding types in the native window system). Developer documentation should mention the correspondence so that developers know what parameters to pass to eglCreateWindowSurface, eglCreatePixmapSurface, and eglCopyBuffers. Documentation should also describe the format of the *display_id* parameter to eglGetDisplay, since this is a platform-specific identifier. See Section 3.2.1, "EGLDisplay" for more details.
- Do not include gl.h in egl.h.

2.1.1.2.3. OpenGL

- For historical reasons both Windows and GNU/Linux include an old version of gl.h, that is beyond the control of Khronos, containing OpenGL 1.2 interfaces. All post-OpenGL 1.2 interfaces of the Compatibility profile of the most recent version of OpenGL are defined in the Khronos-provided glext.h.

[36] http://www.khronos.org/registry/wf/api/1.0/openwfd.1.0.headers.zip
[37] http://www.khronos.org/registry/wf/api/1.0/openwfd.1.0.headers.zip
[38] http://www.pathname.com/fhs/

- `glcorearb.h` defines all the interfaces of the most recent core profile of OpenGL and any ARB extensions to it. It does not include interfaces that have been deprecated and removed from core OpenGL. Vendor extensions to the Core profile should be defined in `glcoreext.h` but implementers should keep in mind that people using the Core profile probably have more concern about portability than most coders and may want to avoid using extensions.

- Implementations supporting only the Core profile should provide the header files listed for OpenGL 3.1+ . Implementations supporting the Compatibility profile should provide the header files listed for OpenGL 3.x & 4.x following the conventions documented in Section 2.2.2, "OpenGL".

- When introducing OpenGL to a platform for the first time, only the Core profile is required. The Compatibility profile is optional and not recommended.

- Khronos recommends using EGL as the window abstraction layer, when introducing OpenGL to a platform. Follow the EGL guidelines given herein.

2.1.1.2.4. OpenGL ES 1.x

- For compatibility with GLES 1.0 implementations, include in `GLES` a special `egl.h`[39] containing the following:

```
#include <EGL/egl.h>
#include <GLES/gl.h>
```

  This is because many early OpenGL ES 1.0 implementations included `gl.h` in `egl.h` so many existing applications only include `egl.h` .

- The name `glu.h` is reserved for future use by the Khronos Group.

2.1.1.2.5. OpenGL ES 2.x

- The names `glu{,2}.h` are reserved for future use by the Khronos Group.

2.1.1.2.6. OpenGL ES 3.x

- The names `glu{,3}.h` are reserved for future use by the Khronos Group.

2.1.1.2.7. OpenKODE

- As noted earlier, *implementers* are strongly encouraged to use the Khronos provided header files. *Implementers*, who are creating their own `kd.h` or need to modify `kdplatform.h`, are urged to code them such that they include as few as possible of the platform's include files, and to avoid declaring C and POSIX standard functions. This will ease the creation of portable OpenKODE applications, and help stop non-portable code being added accidentally.

- Each OpenKODE extension is defined in its own header file. Khronos provided header files for each ratified extension are available in the *Extension Headers* subsection of the OpenKODE registry[40].

2.1.1.2.8. OpenWF

- The OpenWF API is divided into two parts: Composition and Display Control. Separate header files must be provided for each part as indicated in Table 1, "Header File Names and Locations".

[39] http://www.khronos.org/registry/gles/api/1.1/egl.h
[40] http://www.khronos.org/registry/kode/

2.1.2. Libraries

2.1.2.1. Packaging

- It is highly desirable to implement all *API* entry points as function calls. However in OpenKODE Core, macros or in-lines may be used instead of function calls provided the rules in Section 4.3 OpenKODE Core functions[41] of the OpenKODE specification[42] are followed:

  - When calling a function or macro, each argument must be evaluated exactly once (although the order of evaluation is undefined).

  - It must be possible to take the address of a function.

  These rules apply except where individually noted in the specification.

- Except in cases where macros are allowed or versioned symbol naming is recommended (e.g., OpenCL symbol naming), ensure the *API* function names exported by your lib & dll files match the function names specified by the Khronos standard for the *API* you are implementing.

- The entry points for each *API* must be packaged in separate libraries. Recommended library names are given in Table 2, “Recommended Library Names”.

- However to provide backward compatibility for existing applications, two OpenGL ES 1.1 libraries should be provided: one with and one without the EGL entry points.

  Note: There are extant implementations of the dual OpenGL ES libraries demonstrating this is possible on Symbian, GNU/Linux, Win32 and WinCE.

  For OpenGL ES 2.x and 3.x, only a library without EGL entry points is needed.

- To allow interoperability through EGL with other Khronos *APIs*, implementers may want to provide an additional OpenGL library, that can be used with EGL, on platforms having existing GL *ABI* conventions. In such a case, use the name recommended below for the OpenGL library without EGL.

- When introducing OpenGL to a platform, implementers are recommended to provide two OpenGL libraries similarly to the OpenGL ES 1.1 case.

2.1.2.2. Naming

Khronos recommends the library names shown in Table 2, “Recommended Library Names”. The table lists the names developers would typically supply to a link command. The actual file names may vary according to the platform and tools in use.

- The **ld** command on GNU/Linux and Unix systems prefaces the base name with `lib` and appends the extensions `.so` and `.a` in that order when searching for the library. So the full name of the library file must be `lib<base name>.{so,a}` depending on whether it is a shared or static library.

- Neither the Microsoft Visual Studio linker nor the ARM RealView linker offer any special treatment for library names; the extension for a standard library or import library on Windows is `.lib`, while that for a dynamic link library is `.dll`.

- On Windows, if the implementer will provide both 32- and 64-bit libraries, it is necessary to disambiguate by adding a suffix of 32 or 64 to the base name.

[41] http://www.khronos.org/registry/kode/specs/openkode.1.0.2.pdf#nameddest=corefunctions
[42] http://www.khronos.org/registry/kode/specs/openkode.1.0.2.pdf

Khronos therefore recommends that library names be of the form `lib<base name>[<nbits>].<platform specific extension>`; <nbits> being possibly needed only on the Windows platform.

Table 2. Recommended Library Names

API	Part	Base Name
OpenCL		`OpenCL`
EGL		`EGL`
OpenGL 3.x & 4.x	OpenGL[a]	`GL`
	Utilities	`GLU`
OpenGL ES 1.x	Core with EGL (Common Profile)	`GLES_CM`[b c]
	Core with EGL (Lite Profile)	`GLES_CL`[b c]
	Core without EGL	`GLESv1_C[LM]`[d]
	Utilities	`GLUESv1`[e]
OpenGL ES 2.x	Core without EGL	`GLESv2`
	Utilities	`GLUESv2`[e]
OpenGL ES 3.x	Core without EGL	`GLESv2`[f]
	Utilities	`GLUESv2`[e]
OpenKODE	Core	`KD`
OpenMAX AL		`OpenMAXAL`
OpenSL ES		`OpenSLES`
OpenVG	Core	`OpenVG`
	Utilities (when present)	`OpenVGU`
OpenWF	Composition	`WFC`
	Display	`WFD`

[a] May contain Core or Compatibilty profile. See Section 2.2.2, "OpenGL" below.

[b] These names are required for OpenGL ES 1.0 and the libraries must contain the EGL entry points as detailed in Chapter 8, *Packaging*, of the OpenGL ES 1.0 specification.

[c] These names are deprecated for OpenGL ES 1.1 and beyond and should only be used for a library that includes the EGL entry points in order to support legacy applications.

[d] These alternate names for GL ES libraries that do not contain the EGL entry points were introduced starting at revision 1.1.09 of the OpenGL ES 1.1 specification.

[e] These names are reserved for future use by the Khronos Group.

[f] This is not a typo! The v2 name is retained in order that existing applications will continue to run after a system is upgraded to OpenGL ES 3 and avoid a myriad of potential linking issues when developing applications for both v2 & v3.

2.2. Vendor Controlled Platforms (e.g. Android, BREW, iOS, Mac OS X)

2.2.1. General

Vendors of controlled platforms are strongly urged to follow the recommendations given above for Uncontrolled Platforms when adding a Khronos Group *API* to their platform.

Implementers should follow any linkage specifications established by the *platform vendor* .

- Use the header files, (e.g., for OpenGL ES, gl.h, glext.h & egl.h) provided by the *platform vendor*.
- Package header files in the same containing folder used by the *platform vendor*.
- Use the function names specified in those header files.
- Use the function-call convention specified in those header files.
- Implement all *API* entry points in the same way as in the *vendor*-provided *ABI*. That is, functions should be functions, in-line functions should be in-line functions and macros should be macros.
- Use the library names specified by the *platform vendor*.

2.2.2. OpenGL

Because of OpenGL's long history, a series of conventions have been established, even for platforms which are not strictly controlled by their vendors. This section documents the established conventions for all common platforms. They should be followed by Implementers supporting the Compatibility profile in order to provide the least surprises to application developers. Implementers supporting only the Core profile of OpenGL 3.1+ are encouraged to follow the guidelines given in the preceding sections.

Platform *SDKs* typically come with 2 OpenGL-related header files pre-installed: `gl.h`, which defines the OpenGL interfaces, and a header file that defines the interfaces to that platform's window abstraction layer (e.g. `glx.h`) . These files are commonly outdated, often having only OpenGL 1.2 interfaces. As these files are outside the control of Khronos, Khronos provides `*ext.h` files which define newly added functions and enums, such as those specified by OpenGL 4.x. Some platform SDKs may also include `*ext.h` files. These can also be outdated. Implementers are advised to use the Khronos provided `*ext.h` files and supply these with their *SDKs*.

Implementers can choose to provide their own up-to-date `gl.h`. Developer documentation should explain how to ensure the compiler finds any updated or implementation specific files.

With the exceptions noted in the tables below, all files are packaged in the folder `GL`.

The following sections give details for each platform.

In order to allow interaction with other Khronos APIs, implementers may wish to provide an EGL implementation, that supports OpenGL, in addition to a platform's standard window abstraction layer. In this case, EGL should be packaged in a separate library following the guidelines given in Section 2.1, “Uncontrolled Platforms (e.g. GNU®/Linux®, Windows®, Windows CE)” above. Implementers must ensure that an application can link with both the OpenGL library and the EGL library without any errors arising from the OpenGL library having entry points for the platform's standard window abstraction layer.

2.2.2.1. GNU®/Linux® and Unix®

Those implementing OpenGL on GNU/Linux or Unix systems should follow the OpenGL Application Binary Interface for Linux[43] which has been approved by the Linux Standard Base Workgroup[44]. The file and library names given below apply to any platform using the X Window System.

Table 3. GL on GNU/Linux, Unix and other platforms using the X Window System

Abstraction Layer	GLX

[43] http://www.opengl.org/registry/ABI/
[44] http://www.linuxfoundation.org/en/LSB_Workgroup_Home_Page

Header Files	Core	`gl.h`	Mesa via X.org distributions of the X Window System
		`glext.h`[45]	Khronos
	Abstraction Layer	`glx.h`	Mesa via X.org
		`glxext.h`[46]	Khronos
	Utility Library	`glu.h`	Mesa via X.org
Link Library Names	`libGL.so`, `libGLU.so`		
Runtime Library Names	`libGL.so.[134]`,[ab] `libGLU.so.1`		

[a]Per the above referenced ABI, libGL.so.1 contains a minimum of OpenGL 1.2, GLX 1.3 and ARB_multitexture.

[b]The following is subject to change; it will be confirmed at the X.Org Developers Conference in Sep. 2012. libGL.so.[34] should contain only the OpenGL 3.x or 4.x entry points. Applications must link with libEGL for the abstraction layer.

2.2.2.2. Mac® OS X

OpenGL is the primary 3D rendering API for Mac OS X so is a core part of the system. Information is presented here more for completeness than from any expectation that third party implementations will appear. For a complete description of Apple's implementation see the OpenGL Programming Guide for Mac OS X[47]. Primarily aimed at developers, it provides a good description of the structure of OpenGL in Mac OS X.

Apple's SDK includes up-to-date gl.h and glext.h header files that track their implementation. These headers contain all GL 3.x features and are used regardless of the vendor of the underlying GPU. The headers are packaged together with the libraries in the OpenGL Framework.

Table 4. GL on Mac OS X

Abstraction Layers	AGL, CGL, GLX, NSOpenGL		
Header Files	Core Profile	`gl.h`[a]	Apple SDK
		`glext.h`[a]	Apple SDK
	AGL	`agl.h`[b]	Apple SDK
	CGL	`OpenGL.h`[a]	Apple SDK
	`GLX`	`glx.h`[a]	Apple SDK
		`glxext.h`[a]	Apple SDK
	NSOpenGL	`AppKit.framework`	Apple SDK
Library Name	Applications link with the OpenGL framework.		

[a]Packaged in `OpenGL.framework`

[b]Packaged in `AGL.framework`

2.2.2.3. Microsoft Windows®

Those creating an implementation for Windows will need to provide an *Installable Client Driver* (ICD). See Loading an OpenGL Installable Client Driver[48] and OpenGL and Windows Vista[49]™ for more information.

[45] http://www.opengl.org/registry/api/glext.h
[46] http://www.opengl.org/registry/api/glxext.h
[47] http://developer.apple.com/DOCUMENTATION/GraphicsImaging/Conceptual/OpenGL-MacProgGuide/opengl_intro/chapter_1_section_1.html
[48] http://msdn.microsoft.com/en-us/library/aa477537.aspx
[49] http://www.opengl.org/pipeline/article/vol003_7/

Table 5. GL on Microsoft Windows

Abstraction Layer	WGL		
Header Files	OpenGL 1.2	`gl.h`	Microsoft Platform SDK
	All post 1.2 entry points as extensions	`glext.h`[50]	Khronos
	Abstraction Layer	`winGDI.h`[a]	Microsoft Platform SDK
		`wglext.h`[51]	Khronos
	Utility Library	`glu.h`	Microsoft Platform SDK
Library Names	`opengl<nbits>.{lib,dll}`, `glu<nbits>.{lib,dll}`[b]		

[a] `winGDI.h` is a standard Windows' header file so is not packaged in the `GL` folder.

[b] <nbits> is 32 or 64.

3. Implementation Notes

3.1. OpenCL

3.1.1. Offline Compiler

OpenCL implementations may optionally include an off-line compiler. When such a compiler is provided, the compiler must accept the same options as the on-line compiler. These are specified in the OpenCL specification.[52] In addition, the compiler must support a **`-o <file name>`** option for specifying the name of the output file and a **`-b <machine>`** option for specifying the target machine. Khronos recommends that the string "clc" be used as the last part of the compiler name as in, e.g., **openclc** or <**your company name**>**clc**.

3.1.2. Versioned Symbol Naming Convention

OpenCL implementations on GNU/Linux should decorate the symbol names in shared libraries with version numbers in the form `<symbol>@@OPENCL_<version>` where `<version>` is the version of the OpenCL specification in which the symbol first appeared, e.g. 1.0, 1.1 or 2.0.

3.2. EGL

3.2.1. EGLDisplay

The EGL specification[53] (Section 2.1.2 Displays) describes what an EGLDisplay represents. It states that,"In most environments a display corresponds to a single physical screen."In reality most environments have only one EGLDisplay even when multiple physical screens are supported. Many vendors match the EGLDisplay to the display driver implementation. So if there is only one display driver on the system then the system has only one EGLDisplay (even though that one driver may be capable of driving more than 1 physical screen). For example, in the X Window System an EGLDisplay is likely to correspond to an X Display, not to an X Screen.

[50] http://www.opengl.org/registry/api/glext.h
[51] http://www.opengl.org/registry/api/wglext.h
[52] http://www.khronos.org/registry/cl/specs/opencl-1.0.29.pdf
[53] http://www.khronos.org/egl/

The *implementer* is free to choose what an EGLDisplay represents. There could be one EGLDisplay per physical screen. There could be one EGLDisplay per graphics chip or graphics card. There could be one EGLDisplay per graphics driver vendor. Or there could be one EGLDisplay per system which abstracts all available graphics hardware on the system into a single EGLDisplay handle.

While the implementer is free to choose the abstraction for the EGLDisplay, there are advantages to choosing the latter approach where only a single EGLDisplay exists on the system. For example EGLImages and EGLContexts can only be shared within a single EGLDisplay. If there is more than one EGLDisplay on a system (e.g. one per physical screen) it makes sharing resources between the two displays difficult or impossible. Another example is that most applications are written to use a single EGLDisplay (the one corresponding to `EGL_DEFAULT_DISPLAY`), and those apps will not generally be able to take full advantage of systems with multiple EGLDisplays.

Recommendation: Have only one EGLDisplay per system, even if the system has multiple physical screens.

If there is only one EGLDisplay on a system, how does that work with multiple physical screens? This is up to the native window system (or OpenKODE). When an application wants to render to a window on a particular physical screen, it should ensure that the window it creates is displayed on that physical screen. The mechanism is specific to the native window system (or possibly OpenKODE if using `kdCreateWindow()`). NOTE: There is currently no way to tell OpenKODE upon which screen to open a KDWindow. An extension to allow this (by setting a "which screen" window attribute between calling `kdCreateWindow()` and `kdRealizeWindow()`) may be available in the future.

3.2.2. EGLImage

3.2.2.1. Introduction

The various EGLImage specifications describe how EGLImages can be created and used. Some of the implications made by the spec. are not immediately obvious. This section attempts to clarify what choices may need to be made by an implementation, and how an implementation addresses those choices. The discussion in this section assumes that the reader has read at least the EGL_KHR_image_base specification[54].

3.2.2.2. Creation of EGLImages

EGLImages are created with the `eglCreateImageKHR` command. They can be created from a variety of source image objects (e.g. OpenGL ES textures, OpenVG VGImages, native pixmaps, etc).

Regardless of what type of object the EGLImage is created from, the specifications allow an implementation to reallocate the memory which backs the EGLImage. However, there are a number of issues which make this very difficult. For example, when one API is rendering to an EGLImage in one thread, another API is reading the same EGLImage in another thread, and a third thread calls a function which creates a sibling object from the same EGLImage, then it is not clear that reallocation is possible. If an implementation performs reallocation, it is likely that it will have to do expensive locking around the use of EGLImages which will hurt performance. Therefore implementations may benefit from not performing reallocation once an EGLImage has been created.

Recommendation: Implementations should avoid reallocating the memory backing the EGLImage after the EGLImage has been created.

When an EGLImage is created from a client API object, the context that contains the object is current to the thread that called `eglCreateImageKHR()`. Therefore there are fewer issues with reallocating the memory at the time the EGLImage is created than when a sibling is created from the EGLImage.

[54] http://www.khronos.org/registry/egl/extensions/KHR/image_base.txt

Implementations of client APIs may have special requirements for the memory accessed by the API. For example, there may be alignment requirements, or requirements that memory come from a specific range of physical addresses. These requirements may not be the same for all client APIs on a particular system. An OpenGL ES texture may have different alignment requirements than a VGImage, for example.

In addition, there may be constraints on what internal formats are accessible to certain APIs. An implementation may choose to change the internal representation of the image when an EGLImage is created or when a sibling is created from the EGLImage, so long as this is transparent to the application. For example, when an EGLImage is created from an OpenGL ES texture which was specified as RGBA 4444, the implementation may choose to represent the EGLImage internally as an RGBA 8888 image even if the texture was originally internally represented as RGBA 4444. However, in this case the implementation must continue to treat the original texture as an RGBA 4444 texture. This means that `glTexSubImage2D` with *`type`*`=GL_UNSIGNED_BYTE` must fail and `glTexSubImage2D` with *`type`*`=GL_UNSIGNED_SHORT_4_4_4_4` must succeed. An implementation may choose to do this if, for example, its OpenMAX renderer can support RGBA 8888 but not RGBA 4444. Alternatively, an implementation may choose to simply fail to allow an EGLImage represented internally as RGBA 4444 to be used as an OpenMAX buffer.

Generally an implementation may discard the contents of an EGLImage (and all associated client API objects) when the EGLImage is created or when any sibling object is created from the EGLImage. Therefore there is no need to copy and/or convert the contents of a buffer, if the memory backing the EGLImage gets reallocated. Note that, if the EGLImage PRESERVED attribute is set to TRUE when the EGLImage is created, then the contents of the EGLImage (and all associated client API objects) do have to be preserved when the EGLImage is created and when a sibling object is created from that EGLImage. An implementation may still reallocate the memory backing the EGLImage so long as it copies the contents of the old memory to the new memory. An implementation may also opt to fail creation of the EGLImage or the sibling object if the PRESERVED attribute is set to true.

If an implementation chooses not to reallocate the memory backing the EGLImage after it has been created, then it must take extra care when first creating the EGLImage to ensure compatibility among APIs. For example, if the backing memory is allocated using an alignment that is incompatible with an OpenGL ES texture's requirements then that EGLImage cannot be used to create an OpenGL ES texture sibling. An implementation has several choices for addressing this issue:

- Fail to create the sibling object.
- Reallocate the memory with the required alignment.
- Ensure the allocation performed when the EGLImage is created is compatible with all (or as many as possible) types of API objects.

The third choice is the most desirable.

Recommendation: When creating an EGLImage, an implementation should allocate (or reallocate) memory backing the EGLImage in such a way that the memory is compatible with the largest possible set of API objects.

3.2.2.3. Creating EGLImage siblings from EGLImages

As mentioned above, an implementation is free to reallocate memory backing an EGLImage whenever an EGLImage sibling is created from the EGLImage (or at any other time). The only requirement is that any such reallocation be transparent to the application (other than discarding the pixel values, if the PRESERVED attribute is not set to TRUE). As discussed above, performing such reallocation when a sibling is created from an EGLImage may be very difficult to implement robustly. Therefore the recommendation is to avoid such reallocations when creating siblings.

An implementation which never reallocates must fail any attempt to create a sibling from an EGLImage which has an allocation incompatible with the sibling object being created. This failure is acceptable according to the spec, but is still undesirable. This is why it is important for the original memory allocation to be compatible with as many types of API sibling objects as possible.

3.2.2.4. Delayed allocation of EGLImage buffer by siblings at bind time

Another option for allocating memory to back an EGLImage is to delay the allocation until one of the EGLImage sibling objects is actually used. This may allow an implementation to be more flexible and/or perform better. Since the pixels in an EGLImage can be discarded when any sibling object is created, it is advisable for applications using EGLImages to create all sibling objects before using (e.g. rendering to) any of the sibling objects. Delaying the allocation of memory until one of the sibling objects is first used allows the implementation to take into account what type of sibling objects the EGLImage has, and to allocate memory appropriate to all of those sibling objects. Once any of the siblings is actually used (e.g. rendered to) it becomes much more difficult to reallocate the memory.

An application may still request that a sibling object be created from the EGLImage after one of the siblings has been used, but an implementation may choose to fail to create the sibling at this time, if the already allocated memory is incompatible with the requested sibling. Such an implementation would be maximally flexible about what siblings can be created, and maximally efficient in its memory allocation, before any siblings are used. After siblings are used the implementation may be less flexible, but that is a less important case.

3.2.2.5. Performance Notes

Creation of an EGLImage and its sibling objects is not intended to be an operation which is performed frequently (e.g. per frame). It is far more important to optimize the use of EGLImage siblings (binding sibling objects, rendering to sibling objects, and reading from sibling objects (e.g. as textures)) than to optimize their creation and destruction. Where tradeoffs can be made between use performance and creation/destruction performance, it is recommended that the choice be made in favor of optimizing the use of sibling objects even if at the expense of creation/destruction performance (within reason).

3.2.2.6. Developer Documentation

Developer documentation should describe which types of EGLImage can be used for which types of API objects in the implementation. For example, what are the restrictions (if any) on a GLES texture in order for an EGLImage, that is created with it, to be used to create a VGImage sibling from that EGLImage? Are there any additional restrictions which will guarantee optimal performance?

The documentation should also describe which operations are likely to be slow (e.g. creating EGLImages or creating EGLImage siblings).

4. Conformance Testing

To claim your product is compliant with an API specification or use its trademark and logo to market that product, your implementation must pass the API's conformance test and the results must be submitted to the Khronos Group. To do this, you must join Khronos as an Adopter which requires a fee and a legal agreement. More information about the conformance process and fees can be found on the Khronos Adopters page[55]. Adopters will receive an account giving them access to the relevant API-specific Adopters area of the Khronos web site.

No fee or agreement is needed to implement a Khronos API or use it in a product but you may not claim compliance and may not use its trademark and logo.

API-specific Adopters areas provide links for downloading test packages and submitting results. The test packages contain instructions detailing the format and and required content of a Submission Package.

[55] http://www.khronos.org/adopters

A. Glossary

Application Binary Interface (ABI)	The low-level interface between a compiled application program and the operating system or its libraries.
Application Programming Interface (API)	The source-code level interface between an application program and the operating system or its libraries.
Installable Client Driver (ICD)	An OpenGL driver for Microsoft Windows that is identified by a renderer string. The OpenGL runtime decides which ICD to run by reading the contents of a specific registry entry. More details can be found here[1].
Integrated Development Environment (IDE)	A tool for the purpose of software development in which, at a minimum, an editor, compiler and debugger are integrated together for ease of use.
Implementer	A company or person who implements a Khronos API.
Microsoft Foundation Classes (MFC)	A set of C++ utility classes provided by Microsoft Corporation.
Software Development Kit (SDK)	A kit containing the header files, libraries and documentation required to develop software for some device or environment.
Platform Vendor (Vendor)	A company providing an operating system platform that includes an *ABI* specification for one or more Khronos APIs. E.g., Google (OpenGL ES & EGL on Android) and Qualcomm (OpenGL ES on BREW). A Vendor may also be an *Implementer.*

B. Document History

Revision History		
Revision 4.1	2013-12-03	msc
Added note recommending versioned symbol names for OpenCL.		
Revision 4.0	2012-08-06	msc
Fourth Edition. Add info. about OpenGL 4.x and OpenGL ES 3.x.		
Revision 3.0	2009-11-26	msc
Third Edition. Added info. about OpenGL 3.x & OpenWF.		
Revision 2.0	2009-03-05	msc
Second Edition. Reorganized completely; added info. about additional APIs; added Implementation Notes and Conformance Testing sections.		
Revision 1.1	2008-01-11	msc
Added brief note about OpenKODE extensions.		
Revision 1.0	2007-12-28	msc
First Official Release.		

[1] ???

C. Acknowledgements

Thanks to all the members of the Khronos Group for their input and in particular to the following:

Jonathan Grant
Petri Kero
Jon Leech
Ari-Matti Leppänen
Robert Palmer
Acorn Pooley
Jani Vaarala
Hans-Martin Will