The Khronos XML API Registry

Jon Leech

Last updated 2013/09/17

Abstract

This document describes the Khronos XML API Registry schema, and provides some additional information about using the registry and scripts to generate C header files. The underlying XML files and scripts can be obtained starting on the OpenGL.org registry pages at URL

http://www.opengl.org/registry/

Contents

1 Introduction

The Registry is the successor to the ancient `.spec` files used for many years to describe the GL, WGL, and GLX APIs. The `.spec` files had a number of issues including:

- Almost completely undocumented
- Used ancient Perl scripts to read and process the registry.
- Hard to extend and did not semantically capture a variety of things we would like to know about an API.
- Attempted to represent data types using a syntax that bore no description to any actual programming language. Generating this syntax from OpenGL extensions, which describe C bindings, was error-prone and painful for the registry maintainer.
- Could not easily represent related APIs such as OpenGL ES.
- There was an annoying inconsistency about presence of function/token prefixes and vendor suffixes depending on which of the GL, WGL, and GLX `.spec` files was being used.

The new registry uses an XML representation of the API and a set of Python 3 scripts to manipulate the XML, based on the lxml Python bindings. It comes with an XML schema and validator, is somewhat better documented, and we will be much more responsive about updating it.

Some groups outside Khronos have their own XML based API descriptions, often used for additional purposes such as library code generators or extension loaders, and it may be desirable to construct XSLT or other translators between the schema.

2 Downloading the Registry

You can get the processed C header files from the registry pages on the OpenGL.org webserver at URL

http://www.opengl.org/registry/

However, to modify the XML database or the generator scripts for other purposes, you'll need to install a Subversion client and download the registry subrepository at

https://cvs.khronos.org/svn/repos/ogl/trunk/doc/registry/public/api/

3 Getting Started

Once the registry has been obtained from Subversion, if you're running in a Linux command-line environment and have Python 3, the lxml Python bindings, and libxml installed, you should just be able to invoke `make` and generate C/C++ header files for all the following targets:

- `GL/glext.h` - OpenGL 1.2 (and later) compatibility profile API + extensions
- `GL/glcorearb.h` - OpenGL core profile API + extensions
- `GLES/gl.h` - OpenGL compatibility profile API
- `GLES/glext.h` - OpenGL ES 1.x extensions
- `GLES2/gl2.h` - OpenGL ES 2.x API
- `GLES2/gl2ext.h` - OpenGL ES 2.x extensions
- `GLES3/gl3.h` - OpenGL ES 3.x API
- `GL/glx.h` - GLX API
- `GL/glxext.h` - GLX 1.3 (and later) API + extensions
- `GL/wgl.h` - WGL API
- `GL/wglext.h` - WGL extensions
- `EGL/egl.h` - EGL (still being worked on)

Starting with the Makefile rules and inspecting the files `gl.xml`, `genheaders.py`, and `reg.py` will be necessary if you want to repurpose the registry for reasons other than header file generation, or to generate headers for languages other than C.

If you're running in a Windows, MaxOS X, or other environment, there are equivalent versions of Python and GNU Make, although we haven't tested this ourselves. Feedback would be helpful.

3.1 Header Generation Script - `genheaders.py`

When generating header files using the `genheaders.py` script, an API name and profile name are required, as shown in the Makefile examples. Additionally, specific versions and extensions can be required or excluded. Based on this information, the generator script extracts the relevant interfaces and creates a C-language header file for them. `genheaders.py` contains predefined generator options for OpenGL, OpenGL core profile, OpenGL ES 1 / 2 / 3, GLX, and WGL headers.

The generator script is intended to be generalizable to other languages by writing new generator classes. Such generators would have to rewrite the C types and definitions in the XML to something appropriate to their language.

3.2 Registry Processing Script - `reg.py`

Actual XML registry processing is done in `reg.py`, which contains several objects and methods for loading registries and extracting interfaces and extensions for use in header generation. There is some internal documentation in the form of comments although nothing more extensive exists yet, and it's possible the Python scripts will evolve significantly based on public feedback.

4 XML Registry Schema

The format of an XML registry is a top level `<registry>` tag containing `<types>`, `<enums>`, `<commands>`, `<feature>`, and `<extension>` tags describing the different elements of an API, as explained below. This description corresponds to a formal Relax NG schema file, `registry.rnc`, against which the XML registry files can be validated.

At present there are separate registries for:

- OpenGL and OpenGL ES - `gl.xml`
- GLX - `glx.xml`
- WGL - `wgl.xml`
- EGL - `egl.xml` (still in development)

4.1 Profiles

Types and enumerants can have different definitions depending on the API profile requested, which allows us to accomodate minor incompatibilities in the OpenGL and OpenGL ES APIs, for example. Features and extensions can include some elements conditionally depending on the API profile requested.

4.2 API Names

Several tags use a `api` attribute. This is an arbitrary string, specified at header generation time, for labelling properties of a specific API. The string can be, but is not necessarily, an actual API name. As used in `genheaders.py` and `gl.xml`, the API names are `gl`, `gles1`, and `gles2`, corresponding to OpenGL, OpenGL ES 1, and OpenGL ES 2/3, respectively.

5 Registry Root (`<registry>` tag)

A `<registry>` contains the entire definition of one or more related APIs.

5.1 Attributes of `<registry>` tags

None.

5.2 Contents of `<registry>` tags

Zero or more of each of the following tags, normally in this order (although order shouldn't be important):

- `<comment>` - Contains arbitrary text, such as a copyright statement. Unused.
- `<types>` (see section 6) - defines API types. Usually only one tag is used.
- `<groups>` (see section 7) - defines named groups of tokens for possible parameter validation in API bindings for languages other than C. Usually only one tag is used.
- `<enums>` (see section 9) - defines API enumerants (tokens). Usually multiple tags are used.
- `<commands>` (see section 12) - defines API commands (functions). Usually only one tag is used.
- `<feature>` (see section 14) - defines API feature interfaces (API versions, more or less). One tag per feature set.
- `<extensions>` (see section 15) - defines API extension interfaces. Usually only one tag is used, wrapping many extensions.

6 API types (`<types>` tag)

The `<types>` tag contains individual `<type>` tags describing each of the derived types used in the API.

Each `<type>` tag contains legal C code, with attributes or embedded tags denoting the type name.

6.1 Attributes of <type> tags

- requires - another type name this type requires to complete its definition.
- name - name of this type (if not defined in the tag body).
- api - an API name (see <feature> below) which specializes this definition of the named type, so that the same API types may have different definitions for e.g. GL ES and GL.
- comment - arbitrary string (unused).

6.2 Contents of <type> tags

<type> contains text which is legal C code for a type declaration. It may also contain embedded tags:

- <apientry/> - insert a platform calling convention macro here during header generation, used mostly for function pointer types.
- <name> - contains the name of this type (if not defined in the tag attributes).

6.3 Example of a <types> tag

```
<types>
    <type name="stddef"><![CDATA[#include <stddef.h>]]></type>
    <type requires="stddef">typedef ptrdiff_t <name>GLintptr</name>;</type>
</types>
```

If the GLint64 type is required by a command, this will result in the following declarations:

```
#include <stddef.h>
typedef ptrdiff_t GLintptr;
```

7 Enumerant Groups (<groups> tag)

The <groups> tags contain individual <group> tags describing some of the group annotations used for return and parameter types.

7.1 Attributes of <groups> tags

None.

7.2 Contents of <groups> tags

Each <groups> block contains zero or more <group> tags, in arbitrary order (although they are typically ordered by group name, to improve human readability).

7.3 Example of <groups> tags

```
<groups>
    <group name="AccumOp">
<enum name="GL_ACCUM"/>
    </group>

    <group name="AttribMask">
<enum name="GL_ACCUM_BUFFER_BIT"/>
<enum name="GL_ALL_ATTRIB_BITS"/>
    </group>
</groups>
```

8 Enumerant Group (<group> tag)

Each <group> tag defines a single group annotation.

8.1 Attributes of <group> tags

- name - group name, an arbitrary string for grouping a set of enums together within a broader namespace.

8.2 Contents of <group> tags

<group> tags may contain zero or more <enum> tags. Each <enum> tag may contain only a name attribute, which should correspond to a <enum> definition in an <enums> block.

8.3 Meaning of <group> tags

If a <proto> or <param> tag of a <command> has a group attribute defined, and that attribute matches a <group> name, then the return type or parameter type is considered to be constrained to values defined by the corresponding <group>. C language bindings do not attempt to enforce this constraint in any way, but other language bindings may try to do so.

9 Enumerant Blocks (<enums> tag)

The <enums> tags contain individual <enum> tags describing each of the token (enumerant) names used in the API.

9.1 Attributes of <enums> tags

- namespace - a string for grouping many different enums together, currently unused but typically something like GL for all enums in the OpenGL / OpenGL ES shared namespace. Multiple <enums> tags can share the same namespace.
- type - a string describing the data type of the values of this group of enums, currently unused. The only string used at present in the is bitmask.
- start, end - integers defining the start and end of a reserved range of enumerants for a particular vendor or purpose. start must be $\leq$ end. These fields define formal enumerant allocations within a namespace, and are made by the Khronos Registrar on request from implementers following the enum allocation policy.
- vendor - string describing the vendor or purposes to whom a reserved range of enumerants is allocated.
- comment - arbitrary string (unused)

9.2 Contents of <enums> tags

Each <enums> block contains zero or more <enum> and <unused> tags, in arbitrary order (although they are typically ordered by sorting on enumerant values, to improve human readability).

9.3 Example of <enums> tags

```
<enums namespace="AttribMask" type="bitmask">
    <enum value="0x00000001"  name="GL_CURRENT_BIT" />
    <enum value="0x00000002"  name="GL_POINT_BIT" />
</enums>
<enums namespace="GL" start="0x80E0" end="0x810F" vendor="MS">
    <enum value="0x80E0"      name="GL_BGR" />
<unused start="0x80E1" end="0x810F" />
</enums>
```

When processed into a C header, and assuming all these tokens were required, this results in

```
#define GL_CURRENT_BIT 0x00000001
#define GL_POINT_BIT   0x00000001
#define GL_BGR        0x80E0
```

10 Enumerants (<enum> tag)

Each <enum> tag defines a single GL (or other API) token.

10.1 Attributes of <enum> tags

- value - enumerant value, a legal C constant (usually a hexadecimal integer).
- name - enumerant name, a legal C preprocessor token name.
- api - an API name which specializes this definition of the named enum, so that different APIs may have different values for the same token (used to address a few accidental incompatibilities between GL and GL ES).
- type - legal C suffix for the value to force it to a specific type. Currently only u and ull are used, for unsigned 32- and 64-bit integer values, respectively. Separated from the value field since this eases parsing and sorting of values, and is rarely used.
- alias - name of another enumerant this is an alias of, used where token names have been changed as a result of profile changes or for consistency purposes. An enumerant alias is simply a different name for the exact same value. At present, enumerants which are promoted from extension to core API status are not tagged as aliases - just enumerants tagged as aliases in the *Changed Tokens* sections of appendices to the OpenGL Specification. This might change in the future.

10.2 Contents of <enum> tags

<enum> tags have no allowed contents. All information is contained in the attributes.

11 Unused Enumerants (<unused> tag)

Each <unused> tag defines a range of enumerants which is allocated, but not yet assigned to specific enums. This just tracks the unused values and is not needed for header generation.

11.1 Attributes of <unused> tags

- start, end - integers defining the start and end of an unused range of enumerants. start must be $\leq$ end. This range should not exceed the range reserved by the surrounding <enums> tag.
- comment - arbitrary string (unused)

11.2 Contents of <unused> tags

None.

12 Command Blocks (<commands> tag)

The <commands> tag contains definitions of each of the functions (commands) used in the API.

12.1 Attributes of <commands> tags

- namespace - a string defining the namespace in which commands live, currently unused but typically something like GL.

12.2 Contents of <commands> tags

Each <commands> block contains zero or more <command> tags, in arbitrary order (although they are typically ordered by sorting on the command name, to improve human readability).

13 Commands (<command> tag)

The <command> tag contains a structured definition of a single API command (function).

13.1 Attributes of <command> tags

- comment - arbitrary string (unused).

13.2 Contents of <command> tags

- <proto> must be the first element, and is a tag defining the C function prototype of a command as described below, up to the function name but not including function parameters.

- <param> elements for each command parameter follow, defining its name and type, as described below. If a command takes no arguments, it has no <param> tags.

Following these elements, the remaining elements in a <command> tag are optional and may be in any order:

- <alias> has no attributes and contains a string which is the name of another command this command is an alias of, used when promoting a function from extension to ARB or ARB to API status. A command alias describes the case where there are two function names which resolve to the **same** client library code, so (for example) the case where a command is promoted but is also given different GLX protocol would **not** be an alias in this sense.

- <vecequiv> has no attributes and contains a string which is the name of another command which is the *vector equivalent* of this command. For example, the vector equivalent of glVertex3f is glVertex3fv.
- <glx> defines GLX protocol information for this command, as described below. Many GL commands don't have GLX protocol defined, and other APIs such as EGL and WGL don't use GLX at all.

13.3 Command prototype (<proto> tags)

The <proto> tag defines the return type and name of a command.

13.3.1 Attributes of <proto> tags

- group - group name, an arbitrary string.

If the group name is defined, it may be interpreted as described in section 8.3.

13.3.2 Contents of <proto> tags

The text elements of a <proto> tag, with all other tags removed, is legal C code describing the return type and name of a command. In addition it may contain two semantic tags:

- The <ptype> tag is optional, and contains text which is a valid type name found in <type> tag, and indicates that this type must be previously defined for the definition of the command to succeed. Builtin C types, and any derived types which are expected to be found in other header files, should not be wrapped in <ptype> tags.
- The <name> tag is required, and contains the command name being described.

13.4 Command parameter (<param> tags)

The <param> tag defines the type and name of a parameter.

13.4.1 Attributes of <param> tags

- group - group name, an arbitrary string.
- len - parameter length, either an integer specifying the number of elements of the parameter <ptype>, or a complex string expression with poorly defined syntax, usually representing a length that is computed as a combination of other command parameter values, and possibly current GL state as well.

If the group name is defined, it may be interpreted as described in section 8.3.

13.4.2 Contents of <param> tags

The text elements of a <param> tag, with all other tags removed, is legal C code describing the type and name of a function parameter. In addition it may contain two semantic tags:

- The <ptype> tag is optional, and contains text which is a valid type name found in <type> tag, and indicates that this type must be previously defined for the definition of the command to succeed. Builtin C types, and any derived types which are expected to be found in other header files, should not be wrapped in <ptype> tags.
- The <name> tag is required, and contains the command name being described.

13.5 Example of a <commands> tag

```
<commands>
    <command>
<proto>void <name>glBeginConditionalRenderNV</name></proto>
<param><ptype>GLuint</ptype> <name>id</name></param>
<param><ptype>GLenum</ptype> <name>mode</name></param>
<alias name="glBeginConditionalRender" />
<glx type="render" opcode="348" />
    </command>
</commands>
```

When processed into a C header, this results in

```
void glBeginConditionalRenderNV(GLuint id, GLenum mode);
```

14 API Features / Versions (<feature> tag)

API features are described in individual <feature> tags. A feature is the set of interfaces (enumerants and commands) defined by a particular API and version, such as OpenGL 4.0 or OpenGL ES 3.0, and includes all API profiles of that version.

14.1 Attributes of <feature> tags

- api - API name this feature is for (see section 4.2), such as gl or gles2.
- name - version name, used as the C preprocessor token under which the version's interfaces are protected against multiple inclusion. Example: GL_VERSION_4_2.
- protect - an additional preprocessor token used to protect a feature definition. Usually another feature or extension name. Rarely used, for odd circumstances where the definition of a feature or extension requires another to be defined first.

- `number` - feature version number, usually a string interpreted as $majorNumber.minorNumber$. Example: `4.2`.
- `comment` - arbitrary string (unused)

14.2 Contents of `<feature>` tags

Zero or more `<require>` and `<remove>` tags (see section 17), in arbitrary order. Each tag describes a set of interfaces that is respectively required for, or removed from, this feature, as described below.

14.3 Example of a `<feature>` tag

```
<feature api="gl" name="GL_VERSION_3_0" number="3.0">
    <require>
<enum name="GL_COMPARE_REF_TO_TEXTURE" />
<enum name="GL_CLIP_DISTANCE0" />
<command name="glEndTransformFeedback" />
    </require>
    <require profile="compatibility">
<enum name="GL_INDEX" />
    </require>
</feature>
```

When processed into a C header for the `compatibility` profile of OpenGL, this results in (assuming the usual definitions of these GL interfaces):

```
#ifndef GL_VERSION_3_0
#define GL_VERSION_3_0 1
#define GL_COMPARE_REF_TO_TEXTURE   0x884E
#define GL_CLIP_DISTANCE0   0x3000
#define GL_INDEX   0x8222
typedef void (APIENTRYP PFNGLENDTRANSFORMFEEDBACKPROC) (void);
#ifdef GL_GLEXT_PROTOTYPES
GLAPI void APIENTRY glEndTransformFeedback (void);
#endif
#endif /* GL_VERSION_3_0 */
```

If processed into a header for the `core` profile, the definition of `GL_INDEX` would not appear.

15 Extension Blocks (`<extensions>` tag)

The `<extensions>` tag contains definitions of each of the extenions which are defined for the API.

15.1 Attributes of <extensions> tags

None.

15.2 Contents of <extensions> tags

Each <extensions> block contains zero or more <extension> tags, each describing an API extension, in arbitrary order (although they are typically ordered by sorting on the extension name, to improve human readability).

16 API Extensions (<extension> tag)

API extensions are described in individual <extension> tags. An extension is the set of interfaces defined by a particular API extension specification, such as ARB_multitexture. <extension> is similar to <feature>, but instead of having version and profile attributes, instead has a supported attribute, which describes the set of API names which the extension can potentially be implemented against.

16.1 Attributes of <extension> tags

- supported - a regular expression, with an implicit ^ and $ bracketing it, which should match the api tag of a set of <feature> tags.
- protect - an additional preprocessor token used to protect an extension definition. Usually another feature or extension name. Rarely used, for odd circumstances where the definition of an extension requires another to be defined first.
- comment - arbitrary string (unused)

16.2 Contents of <extension> tags

Zero or more <require> and <remove> tags (see section 17), in arbitrary order. Each tag describes a set of interfaces that is respectively required for, or removed from, this extension, as described below.

16.3 Example of an <extensions> tag

```
<extensions>
    <extension name="GL_ARB_robustness" supported="gl|glcore" >
<require>
    <enum name="GL_NO_ERROR" />
    <command name="glGetGraphicsResetStatusARB" />
</require>
<require api="gl" profile="compatibility">
```

```
    <command name="glGetnMapdvARB" />
</require>
    </extension>
</extensions>
```

The `supported` attribute says that the extension can be supported for either the GL compatibility (`gl`) or GL core (`glcore`) API profiles, but not for other APIs. When processed into a C header for the `core` profile of OpenGL, this results in (assuming the usual definitions of these GL interfaces):

```
#ifndef GL_ARB_robustness
#define GL_ARB_robustness 1
#define GL_NO_ERROR   0
typedef GLenum (APIENTRYP PFNGLGETGRAPHICSRESETSTATUSARBPROC) (void);
#ifdef GL_GLEXT_PROTOTYPES
GLAPI GLenum APIENTRY glGetGraphicsResetStatusARB (void);
#endif
#endif /* GL_ARB_robustness */
```

17 Required and Removed Interfaces (<require> and <remove> tags)

A <require> block defines a set of interfaces (types, enumerants and commands) *required* by a <feature> or <extension>. A <remove> block defines a set of interfaces *removed* by a <feature> (this is primarily useful for the OpenGL core profile, which removed many interfaces - extensions should never remove interfaces, although this usage is allowed by the schema). Except for the tag name and behavior, the contents of <require> and <remove> tags are identical.

17.1 Attributes of <require> and <remove> tags

- `profile` - string name of an API profile. Interfaces in the tag are only required (or removed) if the specified profile is being generated. If not specified, interfaces are required (or removed) for all API profiles.
- `comment` - arbitrary string (unused)
- `api` - an API name (see section 4.2). Interfaces in the tag are only required (or removed) if the specified API is being generated. If not specified, interfaces are required (or removed) for all APIs.

  The `api` attribute is only supported inside <extension> tags, since <feature> tags already define a specific API.

17.2 Contents of <require> and <remove> tags

Zero or more of the following tags, in any order:

- <command> specifies an required (or removed) command defined in a <commands> block. The tag has no content, but contains elements:
  - name - name of the command (required).
  - comment - arbitrary string (optional and unused).
- <enum> specifies an required (or removed) enumerant defined in a <enums> block. The tag has no content, but contains elements:
  - name - name of the enumerant (required).
  - comment - arbitrary string (optional and unused).
- <type> specifies a required (or removed) type defined in a <types> block. Most types are picked up implicitly by using the <ptype> tags of commands, but in a few cases, additional types need to be specified explicitly (it is unlikely that a type would ever be removed, although this usage is allowed by the schema). The tag has no content, but contains elements:
  - name - name of the type (required).
  - comment - arbitrary string (optional and unused).

18 General Discussion

18.1 Stability of the XML Database and Schema

The new registry schema, scripts, and databases are evolving in response to feedback and to Khronos' own wishlist. This means the XML schema is subject to change, although most such change will probably be confined to adding attributes to existing tags. The XML databases such as gl.xml will evolve in response to schema changes, to new extensions and API versions, and to general cleanup, such as canonicalization of the XML or sorting of <command> and <extension> tags by name. Changes to the schema will be described in the change log of this document (see section 19). Changes to the .xml files will be described in Subversion revision history.

18.2 Feature Enhancements to the Registry

There are lots of tools and additional tags that would make the XML format more expressive and the tools more capable. Khronos is open to hosting additional processing scripts for other purposes. We're hoping to be much more responsive to bugs filed in the Khronos public bugzilla now that there's a more modern and maintainable framework to build on.

A partial wishlist follows:

- Enhance <command> <alias> tags to describe more relaxed sorts of aliases, such as commands equivalent in behavior and interface, but using different GLX protocol (this might be called a *client-side alias* or something of the sort).

18.3 Type Annotations and Relationship to .spec Files

The initial releases of the XML Registry did not include type annotation and array length information from the old .spec files, which generated a number of complaints from people writing bindings of OpenGL to languages other than C. The majority of these annotations have now been added to the XML, in the form of <group> tags (see section 8) defining groups of related enumerants, group attributes on <proto> and <param> tags specifying that the corresponding return type belongs to a group, and len attributes on <param> tags specifying the array length of the corresponding parameter.

There are many caveats regarding these annotations. For the most part they date from SGI's OpenGL 1.1 implementation, and have not been updated. The defined <group>s therefore do not cover many API parameters which **could** be given a group, and in many cases, the defined <group>s have not been updated to add new enumerants from later versions of OpenGL and OpenGL extensions. The group names are often somewhat misleading (with imbedded vendor extension tags, while the corresponding features have been promoted).

The <group>s added to gl.xml are the enumerant groups defined in enum.spec, and enumext.spec. Many additional group names were used in the annotations in gl.spec, and they fall in several categories discussed below.

18.3.1 Simple API Type Aliases

Group names that were simply an alias for a GL type have been left entirely out of gl.xml, since they offer no useful additional information.

For example, a parameter described as UInt32 in gl.spec is not annotated with group="UInt32" in gl.xml, because this offers no information about the parameter not already present in its declaration as <ptype>GLuint</ptype>.

A few examples of such groups are cl_context, handleARB, Int32, and sync.

18.3.2 Numeric Constraints

Group names representing some type of numerical constraint on a value are retained in <proto> and <param> group attributes, but no definition exists. This is because the existing <group> mechanism can only describe constraints to specific enumerant names.

This is not a regression relative to the .spec files, which also did not describe the meaning of these groups.

A few examples of such groups are CheckedFloat32, ClampedFixed, and CoordD.

18.3.3 GL Object Names

Group names representing an object name obtained from e.g. a `glGen*` command, such as a display list name, are retained in in `<proto>` and `<param> group` attributes, but no definition exists. This is because the existing `<group>` mechanism can only describe constraints to specific enumerant names.

This is not a regression relative to the `.spec` files, which also did not describe the meaning of these groups.

A few examples of such groups are `List` and `Path`.

18.3.4 Groups Not Defined Yet

Group names representing enumerant groups which were not defined in `enum.spec` and `enumext.spec` are retained in `<proto>` and `<param> group` attributes, but no definition exists. Such groups usually are a result of vendors contributing `gl.spec` entries for an extension without contributing corresponding `enum.spec` entries.

This is not a regression relative to the `.spec` files, which also did not describe the meaning of these groups.

A few examples of such groups are `ArrayObjectPNameATI`, `BinormalPointerTypeEXT`, and `SpriteParameterNameSGIX`.

18.3.5 Other Groups

There are probably a few groups which are present in `gl.xml` but should not be. These can be gradually cleaned up over time.

18.3.6 Validating Groups

The `genheaders.py` script has been enhanced to allow validating groups during header generation. Invoking it with the `-validate` option, for example via:

```
$ genheaders.py -validate -registry gl.xml
```

will generate a text dump on standard output showing all parameter type `group` attributes which do **not** have corresponding `<group>` tags.

19 Change Log

- 2013/09/17 - Add `<comment>` attribute to `<command>` tags within `<commands>`.
- 2013/06/24 - Add `<groups>` and `<group>` tags, renamed `<enums>` attribute `class` to `group`, and add parameter type annotation attributes `group` and `len` to both `<proto>` and `<param>`. Add section 18.3 discussing limitations of these annotations. Still need to add examples of the annotation attributes.
- 2013/06/19 - Added `<extensions>` tag block as a wrapper around groups of `<extension>` tags, to ease XML transformations such as sorting extensions by name.

- 2013/06/14 - Expanded description of tags, better formatting
- 2013/06/12 - First release, text file format

Index