The Khronos Vulkan API Registry for Vulkan

Jon Leech

Last updated 2016/02/22

Abstract

This document describes the Khronos Vulkan API Registry schema, and provides some additional information about using the registry and scripts to generate a variety of outputs, including C header files as well as several types of asciidoc include files used in the Vulkan API specification and reference pages. The underlying XML files and scripts are located on the Khronos public Github server as URL

https://github.com/KhronosGroup/Vulkan-Docs

The authoritative copy of the Registry is maintained in the *1.0* branch.

Contents

1 Introduction

The registry uses an XML representation of the Vulkan API, together with a set of Python scripts to manipulate the registry once loaded. The scripts rely on the lxml Python bindings to parse and operate on XML. An XML schema and validator target are included.

The schema is based on, but not identical to that used for the previously published OpenGL, OpenGL ES and EGL API registries. It was extended to represent additional types and concepts not needed for those APIS, such as structure and enumerant types, as well as additional types of registered information specific to Vulkan.

The processed C header file corresponding to the registry is checked in under `src/vulkan/vulkan.h`.

1.1 Schema Choices

The XML schema is not pure XML all the way down. In particular, command return types/names and parameters, and structure members, are described in mixed-mode tag

containing C declarations of the appropriate information, with some XML nodes annotating particular parts of the declaration such as its base type and name. This choice is based on prior experience with the SGI `.spec` file format used to describe OpenCL, and greatly eases human reading and writing the XML, and generating C-oriented output. The cost is that people writing output generators for other languages will have to include enough logic to parse the C declarations and extract the relevant information.

People who don't find the supplied Python scripts to suit their needs are likely to write their own parsers, interpreters, and/or converters operating on the registry XML. We hope that we've provided enough information in this document, the RNC schema (`registry.rnc`), and comments in the Registry (`vk.xml`) itself to enable such projects. If not and you need clarifications; if you have other problems using the registry; or if you have proposed changes and enhancements, then please file issues on Khronos' public Github project at

https://github.com/KhronosGroup/Vulkan-Docs/issues

Please tag your issues with `[Registry]` in the title line to help us categorize them. We expect that we will eventually separate the registry from the specification source into a separate repository, but for now they are mixed together.

2 Getting Started

You will need some tools installed. For Linux systems, you will need the following packages installed (Debian package names are noted, Ubuntu package names are probably the same).

- Python 3.x (`python3`)
- The lxml Python package (`python3-lxml`)
- Libxml (`libxml2`)
- A make tool, such as GNU make

The toolchain has also been run successfully on Microsoft Windows in the Cygwin environment, and the individual Python-based components of the toolchain can be run directly from the Windows command line. More details on this are under src/spec/README in Git.

Once you have the right tools installed, perform the following steps:

- Check out the "vulkan" repository linked above from Khronos Gitlab (there are instructions at the link)
- Go to `vulkan/src/spec` in your checked-out repo
- Invoke `make clobber ; make install`

This should regenerate `vulkan.h` and install it in `../vulkan/vulkan.h`. The result should be identical to the version you just pulled from Gitlab (use `git diff ../vulkan/vulkan.h` to check).

Other Makefile targets include:

- `vulkan-docs` - regenerate the API specification and reference page asciidoc include files in `../doc/specs/vulkan/` under the subdirectories `enums`, `flags`, `protos`, and `structs`. These files are pulled in by the ref pages and API spec to put function prototypes and struct and enum declarations in documentation.

  You should probably **not** update these files if you are proposing an API change - just the `vulkan.h` header. When a git merge is accepted including your change, the docs will be updated at that time. Including the dozens of include files in your merge makes it hard to read and adds no information beyond the header updates. Sometimes it is useful to include just one or two of the include files for context of a global change, however.

  However, you are strongly encouraged to updated the **non-**autogenerated docs under `../doc/specs/vulkan/man/*.txt` to reflect API changes.

- `full_install` - equivalent to `install` followed by `vulkan-docs`.

- `validate` - validate `vk.xml` against the XML schema. Recommended if you're making nontrivial changes.

- `readme.pdf` - regenerate this document from the LaTeX source. Most people will never need to do this. If you do, you must have pdflatex installed, preferably from the TeTeX distribution.

If you just want to modify the API, changing `vk.xml` and running "make" should be all that's needed. See appendix A for some examples of modifying the XML.

If you want to repurpose the registry for reasons other than header file and ref page include generation, or to generate headers for languages other than C, start with the Makefile rules and inspect the files `vk.xml`, `genvk.py`, `reg.py`, and `generator.py`.

If you're using other platforms, merge requests with additional documentation on using the tools on those platforms would be very helpful.

2.1 Header Generation Script - `genvk.py`

When generating header files using the `genvk.py` script, an API name and profile name are required, as shown in the Makefile examples. Additionally, specific API 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. `genvk.py` contains predefined generator options for the current development version of Vulkan 1.0.

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.

2.2 Registry Processing Script - `reg.py`

XML 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.

2.3 Output Generator Script - `generator.py`

Once the registry is loaded, the `COutputGenerator` class defined in `generator.py` is used to create a header file. The `DocOutputGenerator` class is used to create the asciidoc include files. Output generators for other purposes can be added as needed.

3 Vulkan Registry Schema

The format of the Vulkan 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 the only registry in this schema is the core Vulkan API registry, `vk.xml`.

3.1 Profiles

Types and enumerants can have different definitions depending on the API profile requested. This capability is not used in the current Vulkan API but may be in the future. Features and extensions can include some elements conditionally depending on the API profile requested.

3.2 API Names

The schema supports, but does not currently use an `api` attribute on several tags. This is an arbitrary string, specified at header generation time, for labelling properties of a specific API or API profile. The string can be, but is not necessarily, an actual API name. Names starting with "vk" are suggested if and when we start defining profiles of Vulkan.

4 Registry Root (`<registry>` tag)

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

4.1 Attributes of `<registry>` tags

None.

4.2 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.
- <vendorids> (see section 5) - defines Khronos vendor IDs, described in detail in the "Layers and Extensions" appendix of the Vulkan Specification.
- <tags> (see section 6) - defines author prefixes used for extensions and layers. Prefixes are described in detail in the "Layers and Extensions" appendix of the Vulkan Specification.
- <types> (see section 7) - defines API types. Usually only one tag is used.
- <enums> (see section 8) - defines API token names and values. Usually multiple tags are used. Related groups may be tagged as an enumerated type corresponding to a <type> tag, and resulting in a C enum declaration. This ability is heavily used in the Vulkan API.
- <commands> (see section 11) - defines API commands (functions). Usually only one tag is used.
- <feature> (see section 13) - defines API feature interfaces (API versions, more or less). One tag per feature set.
- <extensions> (see section 14) - defines API extension interfaces. Usually only one tag is used, wrapping many extensions.

5 Vendor IDs (<vendorids> tag)

The <vendorids> tag contains individual <vendorid> tags defining vendor IDs for physical devices which do not have PCI vendor IDs.

Each <vendorid> tag contains information defining a single vendor ID.

5.1 Attributes of <vendorid> tags

- name - required. The author prefix, as registered with Khronos. This must match an author prefix in the name field of a <tag> tag.
- id - required. The reserved vendor ID, as a hexadecimal number.
- comment - optional. Arbitrary string (unused).

6 Author Prefixes (<tags> tag)

The <tags> tag contains individual <tag> tags defining each of the reserved author prefixes used by extension and layer authors.

Each <tag> tag contains information defining a single author prefix.

6.1 Attributes of <tag> tags

- name - required. The author prefix, as registered with Khronos. A short, uppercase string, usually an abbreviation of an author, project or company name.
- author - required. The author name, such as a full company or project name.
- contact - required. The contact who registered or is currently responsible for extensions and layers using the prefix, including sufficient contact information to reach the contact such as individual name together with email address, Github username, or other contact information.

7 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 information which can be used to generate C code corresponding to the type. In many cases, this is simply legal C code with attributes or embedded tags denoting the type name and other types used in defining this type. In some cases, additional attribute and embedded type information is used to generate more complicated C types.

7.1 Attributes of <type> tags

- requires - optional. Another type name this type requires to complete its definition.
- name - optional. Name of this type (if not defined in the tag body).
- api - optional. 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. This is unlikely to be used in Vulkan, where a single API supports desktop and mobile devices, but the functionality is retained.
- category - optional. A string which indicates that this type contains a more complex structured definition. At present the only accepted categories are basetype bitmask, define, enum, funcpointer, group, handle, include, struct, and union, as described below.
- comment - optional. Arbitrary string (unused).
- parent only applicable if category is handle. Notes another type with the handle category that acts as a parent object for this type.
- returnedonly only applicable if category is struct or union. Notes that this struct/union is going to be filled in by the API, rather than an application filling it out and passing it to the API.

7.2 Contents of <type> tags

The valid contents depend on the category attribute.

7.2.1 Enumerated types - category "enum"

If the category tag has the value enum, the type is a C enumeration. The body of the tag is ignored in this case. The value of the name attribute must be provided and must match the name attribute of a <enums> tag (see section 8). The enumerant values defined within the <enums> tag are used to generate a C enum type declaration.

7.2.2 Structure types - category "struct" or "union"

If the category tag has the values struct or union, the type is a C structure or union, respectively. In this case, the name attribute must be provided, and the contents of the <type> tag are a series of <member> tags defining the members of the aggregate type, in order, followed by an optional <validity> tag including asciidoc validation language for the structure contents.

Structure member (<member>) tags The <member> tag defines the type and name of a structure or union member.

Attributes of <member> tags

- validextensionstructs - only valid on the pNext member of a struct. This is a comma-separated list of structures that may be accepted by pNext instead of NULL
- len - if the member is an array, len may be one or more of the following things, separated by commas (one for each array indirection): another member of that struct; "null-terminated" for a string; "1" to indicate it's just a pointer (used for nested pointers); or an equation (a LaTeX math expression delimited by latexmath:[$ and $].
- externsync - denotes that the member should be externally synchronized when accessed by Vulkan
- optional - whether this value can be omitted by providing NULL (for pointers), VK_NULL_HANDLE (for handles) or 0 (for bitmasks/values)
- noautovalidity - prevents automatic validity language being generated for the tagged item. Only suppresses item-specific validity - parenting issues etc. are still captured.

Contents of `<member>` **tags** The text elements of a `<member>` tag, with all other tags removed, is a legal C declaration of a struct or union member. In addition it may contain two semantic tags:

- The `<type>` tag is optional. It contains text which is a valid type name found in another `<type>` tag, and indicates that this type must be previously defined for the definition of the command to succeed. Builtin C types should not be wrapper in `<type>` tags.
- The `<name>` tag is required, and contains the struct/union member name being described.

Validation (`<validity>`) tags The `<validity>` tag, if present defines valid use cases and values for structure members.

Contents of `<validity>` **tags** Each `<validity>` tag contains zero or more `<usage>` tags. Each `<usage>` tag is intended to represent a specific validation requirement for the structure and include arbitrary asciidoc text describing that requirement.

7.2.3 All other types

If the `category` attribute is one of `basetype`, `bitfield`, `define`, `funcpointer`, `group`, `handle` or `include`, or is not specified, `<type>` contains text which is legal C code for a type declaration. It may also contain embedded tags:

- `<type>` - nested type tags contain other type names which are required by the definition of this type.
- `<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).

There is no restriction on which sorts of definitions may be made in a given category, although the contents of tags with `category` `enum`, `struct` or `union` are interpreted specially as described above.

However, when generating the header, types within each category are grouped together, and categories are generated in the order given by the following list. Therefore, types in a category should correspond to the intended purpose given for that category. If this recommendation is not followed, it is possible that the resulting header file will not compile due to out-of-order type dependencies.

- `include` (`#include` directives)
- `define` (macro `#define` directives)

- `basetype` (scalar typedefs, such as the definition of `VkFlags`)
- `handle` (invocations of macros defining scalar types such as `VkInstance`)
- `enum` (enumeration types and `#define` for constant values)
- `group` (currently unused)
- `bitmask` (enumeration types whose members are bitmasks)
- `funcpointer` (function pointer typedefs)
- `struct` and `union` together (struct and union types)

7.3 Example of a `<types>` tag

```
<types>
    <type name="stddef">#include &lt;stddef.h&gt;</type>
    <type requires="stddef">typedef ptrdiff_t <name>VKlongint</name>;</type>
    <type name="VkEnum" category="enum"/>
    <type category="struct" name="VkStruct">
        <member><type>VkEnum</type> <name>srcEnum</name></member>
        <member><type>VkEnum</type> <name>dstEnum</name></member>
    </type>
</types>

<enums name="VkEnum" type="enum">
    <enum value="0" name="VK_ENUM_ZERO"/>
    <enum value="42" name="VK_ENUM_FORTY_TWO"/>
</enums>
```

The `VkStruct` type is defined to require the types `VkEnum` and `VKlongint` as well. If `VkStruct` is in turn required by a command or another type during header generation, it will result in the following declarations:

```
#include <stddef.h>
typedef ptrdiff_t VKlongint.

typedef enum {
    VK_ENUM_ZERO = 0,
    VK_ENUM_FORTY_TWO = 42
} VkEnum;

typedef struct {
    VkEnum    dstEnum;
    VkLongint dstVal;
} VkStruct;
```

Note that the angle brackets around `stddef.h` are represented as XML entities in the registry. This could also be done using a CDATA block but unless there are many characters requiring special representation in XML, using entities is preferred.

8 Enumerant Blocks (`<enums>` tag)

The `<enums>` tags contain individual `<enum>` tags describing each of the token names used in the API. In some cases these correspond to a C `enum`, and in some cases they are simply compile-time constants (e.g. `#define`).

8.1 Attributes of `<enums>` tags

- `name` - optional. String naming the C `enum` type whose members are defined by this enum group. If present, this attribute should match the `name` attribute of a corresponding `<type>` tag.
- `type` - optional. String describing the data type of the values of this group of enums. At present the only accepted categories are `enum` and `bitmask`, as described below.
- `start`, `end` - optional. 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, and are made by the Khronos Registrar on request from implementers following the enum allocation policy.
- `vendor` - optional. String describing the vendor or purpose to whom a reserved range of enumerants is allocated.
- `comment` - optional. Arbitrary string (unused).

8.2 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).

8.3 Example of `<enums>` tags

An example showing a tag with attribute `type`="`enum`" is given above in section 7.3. The following example is for non-enumerated tokens.

```
<enums>
    <enum value="256" name="VK_MAX_EXTENSION_NAME"/>
    <enum value="MAX_FLOAT"  name="VK_LOD_CLAMP_NONE"/>
</enums>
```

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

```
#define VK_MAX_EXTENSION_NAME   256
#define VK_LOD_CLAMP_NONE       MAX_FLOAT
```

9 Enumerants (<enum> tag)

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

9.1 Attributes of <enum> tags

- value or bitpos - exactly one of these is allowed and required. value is an enumerant value in the form of a legal C constant (usually a literal decimal or hexadecimal integer, though arbitrary strings are allowed). bitpos is a literal integer bit position in a bitmask.
- name - required. Enumerant name, a legal C preprocessor token name.
- api - optional. An API name which specializes this definition of the named enum, so that different APIs may have different values for the same token. May be used to address a subtle incompatibilities.
- type - optional. Used only when value is specified. 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 value since this eases parsing and sorting of values, and rarely used.
- alias - optional. 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 or bitpos.

9.2 Contents of <enum> tags

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

10 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 for the Registrar's use, and is not used for header generation.

10.1 Attributes of <unused> tags

- start - required, end - optional. Integers defining the start and end of an unused range of enumerants. start must be $\leq$ end. If end is not present, then start defines a single unused enumerant. This range should not exceed the range reserved by the surrounding <enums> tag.

- `vendor` - optional. String describing the vendor or purposes to whom a reserved range of enumerants is allocated. Usually identical to the `vendor` attribute of the surrounding `enums` block.
- `comment` - optional. Arbitrary string (unused).

10.2 Contents of `<unused>` tags

None.

11 Command Blocks (`<commands>` tag)

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

11.1 Attributes of `<commands>` tags

None.

11.2 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).

12 Commands (`<command>` tag)

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

12.1 Attributes of `<command>` tags

- `queues` - optional. A string identifying the command queues this command can be placed on. The format of the string is one or more of the terms `"compute"`, `"dma"`, and `"graphics"`, with multiple terms separated by commas (`","`).
- `successcodes` - optional. A string describing possible successful return codes from the command, as a comma-separated list of Vulkan result code names.
- `errorcodes` - optional. A string describing possible error return codes from the command, as a comma-separated list of Vulkan result code names.
- `renderpass` - optional. A string identifying whether the command can be issued only inside a render pass (`"inside"`), only outside a render pass (`"outside"`), or both (`"both"`).

- `cmdbufferlevel` - optional. A string identifying the command buffer levels that this command can be called by. The format of the string is one or more of the terms `"primary"` and `"secondary"`, with multiple terms separated by commas (`","`).
- `comment` - optional. Arbitrary string (unused).

12.2 Contents of `<command>` tags

- `<proto>` is required and must be the first element. It is a tag defining the C function prototype of a command as described below, up to the function name and return type 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.
- An optional `<validity>` tag including asciidoc validation language for the command parameters.

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

- `<alias>` - optional. 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 vendor to Khronos extension or Khronos extension to core API status. A command alias describes the case where there are two function names which resolve to the **same** entry point in the underlying layer stack.
- `<description>` - optional. Unused text.
- `<implicitexternsyncparams>` - optional. Contains a list of `<param>` tags, each containing Asciidoc source text describing an object which is not a parameter of the command, but is related to one, and which also requires external synchronization as described in section 12.4.1. The text is intended to be incorporated into the API specification.

12.3 Command prototype (`<proto>` tags)

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

12.3.1 Attributes of `<proto>` tags

None.

12.3.2 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 to text, it may contain two semantic tags:

- The <type> tag is optional, and contains text which is a valid type name found in a <type> tag. It 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 <type> tags.
- The <name> tag is required, and contains the command name being described.

12.4 Command parameter (<param> tags)

The <param> tag defines the type and name of a parameter. Its contents are very similar to the <member> tag used to define struct and union members.

12.4.1 Attributes of <param> tags

- len - if the param is an array, len may be one or more of the following things, separated by commas (one for each array indirection): another param of that command; "null-terminated" for a string; "1" to indicate it's just a pointer (used for nested pointers); or an equation (a simple expression prefixed with "math:")
- optional - whether this value can be omitted by providing NULL (for pointers), VK_NULL_HANDLE (for handles) or 0 (for bitmasks/values)
- noautovalidity - prevents automatic validity language being generated for the tagged item. Only suppresses item-specific validity - parenting issues etc. are still captured.
- externsync - optional. A boolean string, which must have the value "true" if present, indicating that this parameter (e.g. the object a handle refers to, or the contents of an array a pointer refers to) is modified by the command, and is not protected against modification in multiple app threads. Parameters which do not have this attribute are assumed to not require external synchronization.

12.4.2 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 <type> 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 <type> tags.

- The <name> tag is required, and contains the parameter name being described.

12.5 Example of a <commands> tag

```
<commands>
    <command>
        <proto><type>VkResult</type> <name>vkCreateInstance</name></proto>
        <param>const <type>VkInstanceCreateInfo</type>* <name>pCreateInfo</name>
        <param><type>VkInstance</type>* <name>pInstance</name></param>
    </command>
</commands>
```

When processed into a C header, this results in

```
VkResult vkCreateInstance(
    const VkInstanceCreateInfo* pCreateInfo,
    VkInstance* pInstance);
```

12.6 Parameter validation (<validity>) tags

The <validity> tag, if present defines valid use cases and values for command parameters

Contents of <validity> tags Each <validity> tag contains zero or more <usage> tags. Each <usage> tag is intended to represent a specific validation requirement for the command, and contains arbitrary asciidoc text describing that requirement.

13 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 Vulkan 1.0, and includes all profiles of that API and version.

13.1 Attributes of <feature> tags

- api - required. API name this feature is for (see section 3.2), such as vk.
- name - required. Version name, used as the C preprocessor token under which the version's interfaces are protected against multiple inclusion. Example: VK_VERSION_1_0.

- `name` - required. Version name, used as the C preprocessor token under which the version's interfaces are protected against multiple inclusion. Example: `VK_VERSION_1_0`.
- `number` - required. Feature version number, usually a string interpreted as $majorNumber.minorNumber$. Example: `4.2`.
- `protect` - optional. 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.
- `comment` - optional. Arbitrary string (unused).

13.2 Contents of `<feature>` tags

Zero or more `<require>` and `<remove>` tags (see section 16), in arbitrary order. Each tag describes a set of interfaces that is respectively required for, or removed from, this feature, as described below.

13.3 Example of a `<feature>` tag

```
<feature api="vulkan" name="VK_VERSION_1_0" number="1.0">
    <require comment="Header boilerplate">
        <type name="vk_platform"/>
    </require>
    <require comment="API constants">
        <enum name="VK_MAX_PHYSICAL_DEVICE_NAME"/>
        <enum name="VK_LOD_CLAMP_NONE"/>
    </require>
    <require comment="Device initialization">
        <command name="vkCreateInstance"/>
    </require>
</feature>
```

When processed into a C header for Vulkan, this results in:

```
#ifndef VK_VERSION_1_0
#define VK_VERSION_1_0 1
#define VK_MAX_EXTENSION_NAME   256
#define VK_LOD_CLAMP_NONE       MAX_FLOAT
typedef VkResult (VKAPI_PTR *PFN_vkCreateInstance)(const VkInstanceCreateInfo* p
#ifdef VK_PROTOTYPES
VKAPI_ATTR VkResult VKAPI_CALL vkCreateInstance(
    const VkInstanceCreateInfo*                 pCreateInfo,
    VkInstance*                                 pInstance);
#endif
#endif /* VK_VERSION_1_0 */
```

14 Extension Blocks (<extensions> tag)

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

14.1 Attributes of <extensions> tags

None.

14.2 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).

15 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.

15.1 Attributes of <extension> tags

- name - required. Extension name, following the conventions in the Vulkan Specification. Example: name="VK_VERSION_1_0".
- number - required. A decimal number which is the registered, unique extension number for name.
- supported - required. A regular expression, with an implicit ^ and $ bracketing it, which should match the api tag of a set of <feature> tags.
- protect - optional. 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 extension or a header file to be defined first.
- author - optional. The author name, such as a full company name. If not present, this can be taken from the corresponding <tag> attribute. However, EXT and other multi-vendor extensions may not have a well-defined author or contact in the tag.

- `contact` - optional. The contact who registered or is currently responsible for extensions and layers using the tag, including sufficient contact information to reach the contact such as individual name together with email address, Github username, or other contact information. If not present, this can be taken from the corresponding `<tag>` attribute just like `author`.
- `comment` - optional. Arbitrary string (unused).

15.2 Contents of `<extension>` tags

Zero or more `<require>` and `<remove>` tags (see section 16), in arbitrary order. Each tag describes a set of interfaces that is respectively required for, or removed from, this extension, as described below.

15.3 Example of an `<extensions>` tag

```
<extension name="VK_KHR_display_swapchain" number="4" supported="vulkan">
    <require>
        <enum value="9" name="VK_KHR_DISPLAY_SWAPCHAIN_SPEC_VERSION"/>
        <enum value="4" name="VK_KHR_DISPLAY_SWAPCHAIN_EXTENSION_NUMBER"/>
        <enum value="&amp;quot;VK_KHR_display_swapchain&amp;quot;"
              name="VK_KHR_DISPLAY_SWAPCHAIN_EXTENSION_NAME"/>
        <type name="VkDisplayPresentInfoKHR"/>
        <command name="vkCreateSharedSwapchainsKHR"/>
    </require>
</extension>
```

The `supported` attribute says that the extension is defined for the default profile (`vulkan`). When processed into a C header for the `vulkan` profile, this results in header contents something like (assuming corresponding definitions of the specified `<type>` and `<command>` elsewhere in the XML):

```
#define VK_KHR_display_swapchain 1
#define VK_KHR_DISPLAY_SWAPCHAIN_SPEC_VERSION 9
#define VK_KHR_DISPLAY_SWAPCHAIN_EXTENSION_NUMBER 4
#define VK_KHR_DISPLAY_SWAPCHAIN_EXTENSION_NAME "VK_KHR_display_swapchain"

typedef struct VkDisplayPresentInfoKHR {
    VkStructureType                                 sType;
    const void*                                     pNext;
    VkRect2D                                        srcRect;
    VkRect2D                                        dstRect;
    VkBool32                                        persistent;
} VkDisplayPresentInfoKHR;

typedef VkResult (VKAPI_PTR *PFN_vkCreateSharedSwapchainsKHR)(
    VkDevice device, uint32_t swapchainCount,
```

```
    const VkSwapchainCreateInfoKHR* pCreateInfos,
    const VkAllocationCallbacks* pAllocator,
    VkSwapchainKHR* pSwapchains);

#ifdef VK_PROTOTYPES
VKAPI_ATTR VkResult VKAPI_CALL vkCreateSharedSwapchainsKHR(
    VkDevice                                    device,
    uint32_t                                    swapchainCount,
    const VkSwapchainCreateInfoKHR*             pCreateInfos,
    const VkAllocationCallbacks*                pAllocator,
    VkSwapchainKHR*                             pSwapchains);
#endif
```

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

A <require> block defines a set of interfaces (types, enumerants and commands) and additional validity statements *required* by a <feature> or <extension>. A <remove> block defines a set of interfaces or validity statements *removed* by a <feature> This is primarily for future profiles of an API which may choose to deprecate and/or remove some interfaces - or for profiles or extensions to remove validity statements. 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.

16.1 Attributes of <require> and <remove> tags

- profile - optional. 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 - optional. Arbitrary string (unused).
- api - optional. An API name (see section 3.2). 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.

16.2 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 attributes:

  - `name` - required. Name of the command.
  - `comment` - optional. Arbitrary string (unused).

- `<enum>` specifies an required (or removed) enumerant defined in a `<enums>` block. All forms of this tag support the following attributes:

  - `name` - required. Name of the enumerant.
  - `comment` - optional. Arbitrary string (unused).

  There are two forms of `<enum>` tags:

  Reference enums simply pull in the definition of an enumerant given in a separate `<enums>` block. Reference enums are the most common usage, and no attributes other than `name` and `comment` are supported for them.

  Extension enums define the value of an enumerant inline in an `<extensions>` block. There are several variants, depending on which additional tags are defined:

  - Attributes `value` and (optionally) `type` define a constant value in the same fashion as an `<enum>` tag in an `<enums>` block (see section 9).
  - Attribute `bitpos` defines a constant bitmask value in the same fashion as an `<enum>` tag in an `<enums>` block (see section 9).
  - Attribute `extends` allows an extension enumerant to be added to a separately defined enumerated type whose name is specified by the contents of the `extends` attribute (e.g. a `<type>` tag with `category` "enum", pulling in an `<enums>` block).

    There are two ways to extend an enumerated type:

    * If `bitpos` is also specified, the tag defines a bitmask value, but adds its definition to the enumeranted type specified by `extends` instead of as a compile-time constant.
    * If `offset` is also specified, the tag adds a new enumerant value to the enumerated type specified by `extends`. The actual value defined depends on the extension number (the `number` attribute of the `<extensions>` tag) and the offset, as defined in the "Layers and Extensions" appendix of the Vulkan Specification. The `dir` attribute may also be specified as `dir="-"` if the calculated value should be negative instead of positive. Negative enumerant values are normally used only for Vulkan error codes.

  Examples of various types of extension enumerants are given below in section 16.3.

- `<type>` specifies a required (or removed) type defined in a `<types>` block. Most types are picked up implicitly by using the `<type>` 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` - required. Name of the type.
  - `comment` - optional. Arbitrary string (unused).

- `<usage>` specifies a required (or removed) validity statement for a `<validity>` block belonging to a `<proto>` or `<type>` block. Validity is more likely to be removed by extensions than other items. One of struct or command must be present.
  - `struct` - optional. Name of the structure this validity is added to (or removed from).
  - `comment` - optional. Name of the command this validity is added to (or removed from).

16.3 Examples of Extension Enumerants

Examples of each of the supported extension enumerant `<enum>` tags are given below. Note that extension enumerants are supported only inside `<extension>` blocks - not in `<feature>` blocks[1].

```
<extensions>
    <extension name="VK_KHR_test_extension" number="1" supported="vulkan">
        <require>
            <enum value="42" name="VK_KHR_theanswer"/>
            <enum bitpos="29" name="VK_KHR_bitmask"/>
            <enum offset="0" dir="-" extends="VkResult"
                  name="VK_ERROR_SURFACE_LOST_KHR"/>
            <enum offset="1" extends="VkResult"
                  name="VK_SUBOPTIMAL_KHR"/>
            <enum bitpos="31" extends="VkResult"
                  name="VK_KHR_EXTENSION_BITFIELD"/>
        </require>
    </extension>
</extensions>
```

The corresponding header file will include definitions like this:

```
typedef enum VkResult {
    <previously defined VkResult enumerant values},
    VK_ERROR_SURFACE_LOST_KHR = -1000000000,
    VK_SUBOPTIMAL_KHR = 1000000001,
    VK_KHR_EXTENSION_BITFIELD = 0x80000000,
};

#define VK_KHR_test_extension 1
```

[1] However, we will have to define additional XML tags and/or syntax for future core versions of Vulkan, to properly tag and group core enumerants for Vulkan 1.0, 1.1, etc.

```
#define VK_KHR_theanswer 42
#define VK_KHR_bitmask 0x20000000
```

A Examples / FAQ / How Do I?

For people new to the Registry, it won't be immediately obvious how to make changes. This section includes some tips and examples that will help you make changes to the Vulkan headers by changing the Registry XML description.

First, follow the steps in section 2 to get the Vulkan Gitlab repository containing the registry and assemble the tools necessary to work with the XML registry. Once you're able to regenerate `vulkan.h` from `vk.xml`, you can start making changes.

A.1 General Strategy

If you are **adding** to the API, perform the following steps to **create** the description of that API element:

- For each type, enum group, compile-time constant, and command being added, create appropriate new `<type>`, `<enums>`, `<enum>`, or `<command>` tags defining the interface in question.
- Make sure that all added types and commands appropriately tag their dependencies on other types by adding nested `<type>` tags.
- Make sure that each new tag defines the name of the corresponding type, enum group, constant, or command, and that structure/union types and commands tag the types and names of all their members and parameters. This is essential for the automatic dependency process to work.

If you are **modifying** existing APIs, just make appropriate changes in the existing tags.

Once the definition is added, proceed to the next section to create dependencies on the changed feature.

A.2 API Feature Dependencies

When you add new API elements, they will not result in corresponding changes in the generated header unless they are **required** by the interface being generated. This makes it possible to include different API versions and extensions in a single registry and pull them out as needed. So you must introduce a dependency on new features in the corresponding `<feature>` tag.

Initially, the only API feature is Vulkan 1.0, so there is only one `<feature>` tag in `vk.xml`. You can find it by searching for the following block of `vk.xml`:

```
<!-- SECTION: Vulkan API interface definitions -->
<feature api="vulkan" name="VK_VERSION_1_0" number="1.0">
```

Inside the `<feature>` tag are nested multiple `<require>` tags. These are just being used as a logical grouping mechanism for related parts of Vulkan 1.0 at present, though they may have more meaningful roles in the future if different API profiles are defined.

A.2.1 API Feature Walkthrough

This section walks through the first few required API features in the `vk.xml` `<feature>` tag, showing how each requirement pulls in type, token, and command definitions and turns those into definitions in the C header file `vulkan.h`.

Consider the first few lines of the `<feature>`:

```
<require comment="Header boilerplate">
    <type name="vk_platform"/>
</require>
<require comment="API constants">
    <enum name="VK_MAX_PHYSICAL_DEVICE_NAME"/>
    <enum name="VK_MAX_EXTENSION_NAME"/>
    ...
</require>
<require comment="Device initialization">
    <command name="vkCreateInstance"/>
    ...
```

The first `<require>` block says to require a type named `vk_platform`. If you look at the beginning of the `<types>` section, there's a corresponding definition:

```
<type name="vk_platform">#include "vk_platform.h"
#define VK_MAKE_VERSION(major, minor, patch) \
    ((major &lt;&lt; 22) | (minor &lt;&lt; 12) | patch)
    ...
```

section which is invoked by the requirement and emits a bunch of boilerplate C code. The explicit dependency isn't strictly required since `vk_platform` will be required by many other types, but placing it first causes this to appear first in the output file.

Note that `vk_platform` does not correspond to an actual C type, but instead to a collection of freeform preprocessor includes and macros and comments. Most other `<type>` tags do define a specific type and are much simpler, but this approach can be used to inject arbitrary C into `vulkan.h` **when there's no other way**. In general inserting arbitrary C is strongly discouraged outside of specific special cases like this.

The next `<require>` block pulls in some compile-time constants. These correspond to the definitions found in the first `<enums>` section of `vk.xml`:

```
<!-- SECTION: Vulkan enumerant (token) definitions. -->

<enums comment="Misc. hardcoded constants - not an enumerated type">
        <!-- This is part of the header boilerplate -->
    <enum value="256"        name="VK_MAX_PHYSICAL_DEVICE_NAME"/>
    <enum value="256"        name="VK_MAX_EXTENSION_NAME"/>
    ...
```

The third `<require>` block starts pulling in some Vulkan commands. The first command corresponds to the following definition found in the `<commands>` section of `vk.xml`:

```
<commands>
    <command>
        <proto><type>VkResult</type> <name>vkCreateInstance</name></proto>
        <param>const <type>VkInstanceCreateInfo</type>* <name>pCreateInfo</name>
        <param><type>VkInstance</type>* <name>pInstance</name></param>
    </command>
    ...
```

In turn, the `<command>` tag requires the `<type>`s `VkResult`, `VkInstanceCreateInfo`, and `VkInstance` as part of its definition. The definitions of these types are determined as follows:

For `VkResult`, the corresponding required `<type>` is:

```
<type name="VkResult" category="enum"/>
```

Since this is an enumeration type, it simply links to an `<enums>` tag with the same name:

```
<enums name="VkResult" type="enum" comment="Error and return codes">
    <!-- Return codes for successful operation execution (positive values) -->
    <enum value="0"     name="VK_SUCCESS"/>
    <enum value="1"     name="VK_UNSUPPORTED"/>
    <enum value="2"     name="VK_NOT_READY"/>
    ...
```

For `<VkInstanceCreateInfo>`, the required `<type>` is:

```
<type category="struct" name="VkInstanceCreateInfo">
    <member><type>VkStructureType</type>        <name>sType</name></member>
    <member>const void*                         <name>pNext</name></member>
    <member>const <type>VkApplicationInfo</type>* <name>pAppInfo</name></member>
    <member>const <type>VkAllocCallbacks</type>* <name>pAllocCb</name></member>
    <member><type>uint32_t</type>               <name>extensionCount</name></mem
    <member>const <type>char</type>*const*      <name>ppEnabledExtensionNames</n
</type>
```

This is a structure type, defining a C `struct` with all the members defined in each `<member>` tag in order. In addition, it requires some other types, whose definitions are located by name in exactly the same fashion.

For the final direct dependency of the command, `VkInstance`, the required `<type>` is:

```
<!-- Types which can be void pointers or class pointers, selected at compile tim
<type>VK_DEFINE_BASE_HANDLE(<name>VkObject</name>)</type>
<type>VK_DEFINE_DISP_SUBCLASS_HANDLE(<name>VkInstance</name>, <type>VkObject</ty
```

In this case, the type `VkInstance` is defined by a special compile-time macro which defines it as a derived class of `VkObject` (for C++) or a less typesafe definition for (for C). This macro isn't part of the type dependency analysis, just the boilerplate used in the header.

If these are the only `<feature>` dependencies in `vk.xml`, the resulting `vulkan.h` header will look like this:

```
#ifndef VULKAN_H_
#define VULKAN_H_ 1

#ifdef __cplusplus
extern "C" {
#endif

/*
** Copyright (c) 2015-2016 The Khronos Group Inc.
    ...
*/

/*
** This header is generated from the Khronos Vulkan XML API Registry.
**
** Generated on date 20160208
*/


#define VK_VERSION_1_0 1
#include "vk_platform.h"
#define VK_MAKE_VERSION(major, minor, patch) \
    ((major << 22) | (minor << 12) | patch)

// Vulkan API version supported by this file
#define VK_API_VERSION VK_MAKE_VERSION(0, 104, 0)

#if defined (__cplusplus) && (VK_UINTPTRLEAST64_MAX == UINTPTR_MAX)
    #define VK_TYPE_SAFE_COMPATIBLE_HANDLES 1
#endif

#if defined(VK_TYPE_SAFE_COMPATIBLE_HANDLES) && !defined(VK_DISABLE_TYPE_SAFE_HA
    #define VK_DEFINE_PTR_HANDLE(_obj) struct _obj##_T { char _dummy; }; typedef
    #define VK_DEFINE_PTR_SUBCLASS_HANDLE(_obj, _base) struct _obj##_T : public

    #define VK_DEFINE_BASE_HANDLE(_obj) VK_DEFINE_PTR_HANDLE(_obj)
    #define VK_DEFINE_DISP_SUBCLASS_HANDLE(_obj, _base) VK_DEFINE_PTR_SUBCLASS_H
    #define VK_DEFINE_NONDISP_SUBCLASS_HANDLE(_obj, _base) VK_DEFINE_PTR_SUBCLAS
```

```
#else
    #define VK_DEFINE_BASE_HANDLE(_obj) typedef VkUintPtrLeast64 _obj;
    #define VK_DEFINE_DISP_SUBCLASS_HANDLE(_obj, _base) typedef uintptr_t _obj;
    #define VK_DEFINE_NONDISP_SUBCLASS_HANDLE(_obj, _base) typedef VkUintPtrLeas
#endif

typedef enum {
    VK_SUCCESS = 0,
    VK_UNSUPPORTED = 1,
    VK_NOT_READY = 2,
    ...
} VkResult;
typedef enum {
    VK_STRUCTURE_TYPE_APPLICATION_INFO = 0,
    ...
} VKStructureType;
typedef struct {
    VkStructureType                             sType;
    const void*                                 pNext;
    const char*                                 pAppName;
    uint32_t                                    appVersion;
    const char*                                 pEngineName;
    uint32_t                                    engineVersion;
    uint32_t                                    apiVersion;
} VkApplicationInfo;
typedef enum {
    VK_SYSTEM_ALLOC_TYPE_API_OBJECT = 0,
    ...
} VkSystemAllocType;
typedef void* (VKAPI_PTR *PFN_vkAllocFunction)(
    void*                           pUserData,
    size_t                          size,
    size_t                          alignment,
    VkSystemAllocType               allocType);
typedef void (VKAPI_PTR *PFN_vkFreeFunction)(
    void*                           pUserData,
    void*                           pMem);
typedef struct {
    void*                                       pUserData;
    PFN_vkAllocFunction                         pfnAlloc;
    PFN_vkFreeFunction                          pfnFree;
} VkAllocCallbacks;
typedef struct {
    VkStructureType                             sType;
    const void*                                 pNext;
    const VkApplicationInfo*                    pAppInfo;
```

```
    const VkAllocCallbacks*                     pAllocCb;
    uint32_t                                    extensionCount;
    const char*const*                           ppEnabledExtensionNames;
} VkInstanceCreateInfo;
VK_DEFINE_BASE_HANDLE(VkObject)
VK_DEFINE_DISP_SUBCLASS_HANDLE(VkInstance, VkObject)
#define VK_MAX_PHYSICAL_DEVICE_NAME       256
#define VK_MAX_EXTENSION_NAME             256
typedef VkResult (VKAPI_PTR *PFN_vkCreateInstance)(const VkInstanceCreateInfo* p
#ifdef VK_PROTOTYPES
VKAPI_ATTR VkResult VKAPI_CALL vkCreateInstance(
    const VkInstanceCreateInfo*                 pCreateInfo,
    VkInstance*                                 pInstance);
#endif

#ifdef __cplusplus
}
#endif

#endif
```

Note that several additional types are pulled in by the type dependency analysis, but only those types, commands, and tokens required by the specified features are generated.

A.3 How To Add A Compile-Time Constant

Go to the `<feature>` tag and search for the nested block labelled

```
<require comment="API constants">
```

In this block, add an (appropriately indented) tag like

```
  <enum name="VK_THE_ANSWER"/>
```

Then go to the `<enums>` block labelled

```
<enums comment="Misc. hardcoded constants - not an enumerated type">
```

In this block, add a tag whose `name` attribute matches the `name` you defined above and whose `value` attribute is the value to give the constant:

```
    <enum value="42"    name="VK_THE_ANSWER"/>
```

A.4 How To Add A Struct or Union Type

For this example, assume we want to define a type corresponding to a C `struct` defined as follows:

```
typedef struct {
    VkStructureType          sType;
    const void*              pNext;
    const VkApplicationInfo* pAppInfo;
    const VkAllocCallbacks*  pAllocCb;
    uint32_t                 extensionCount;
    const char*const*        ppEnabledExtensionNames;
} VkInstanceCreateInfo;
```

If `VkInstanceCreateInfo` is the type of a parameter of a command in the API, make sure that command's definition (see below for how to add a command) puts `VkInstanceCreateInfo` in nested `<type>` tags where it's used.

Otherwise, if the struct type is not used directly by a command in the API, nor required by a chain of type dependencies for other commands, an explicit `<type>` dependency should be added to the `<feature>` tag. Go to the `<types>` tag and search for the nested block labelled

```
<require comment="API types not used by commands">
    ...
```

In this block, add a tag whose `name` attribute matches the `name` of the struct type being defined:

```
<require comment="API types not used by commands">
    <type name="VkInstanceCreateInfo"/>
    ...
```

Then go to the `<types>` tag and add a new `<type>` tag defining the struct names and members, somewhere below the corresponding comment, like this:

```
<types>
    ...
    <!-- Struct types -->
        <type category="struct" name="VkInstanceCreateInfo">
            <member><type>VkStructureType</type>
                    <name>sType</name></member>
            <member>const void*
                    <name>pNext</name></member>
            <member>const <type>VkApplicationInfo</type>*
                    <name>pAppInfo</name></member>
            <member>const <type>VkAllocCallbacks</type>*
                    <name>pAllocCb</name></member>
            <member><type>uint32_t</type>
```

```
                <name>extensionCount</name></member>
            <member>const <type>char</type>*const*
                <name>ppEnabledExtensionNames</name></member>
        </type>
        ...
```

If any of the member types are types also defined in the header, make sure to enclose those type names in nested `<type>` tags, as shown above. Basic C types should not be tagged.

If the type is a C `union`, rather than a `struct`, then set the value of the `category` attribute to `"union"` instead of `"struct"`.

A.5 How To Add An Enumerated Type

For this example, assume we want to define a type corresponding to a C `enum` defined as follows:

```
typedef enum {
    VK_DEVICE_CREATE_VALIDATION_BIT = 0x00000001,
    VK_DEVICE_CREATE_MULTI_DEVICE_IQ_MATCH_BIT = 0x00000002;
} VkDeviceCreateFlagBits.
```

If `VkDeviceCreateFlagBits` is the type of a parameter to a command in the API, or of a member in a structure or union, make sure that command parameter or struct member's definition puts `VkDeviceCreateFlagBits` in nested `<type>` tags where it's used.

Otherwise, if the enumerated type is not used directly by a command in the API, nor required by a chain of type dependencies for commands and structs, an explicit `<type>` dependency should be added to the `<feature>` tag in exactly the same fashion as described above for `struct` types.

Next, go to the line labelled

```
<!-- SECTION: Vulkan enumerant (token) definitions. -->
```

Below this line, add an `<enums>` tag whose `name` attribute matches the `<type>` name `VkDeviceCreateFlagBits`, and whose contents correspond to the individual fields of the enumerated type:

```
<enums name="VkDeviceCreateFlagBits" type="bitmask">
    <enum bitpos="0" name="VK_DEVICE_CREATE_VALIDATION_BIT"/>
    <enum bitpos="1" name="VK_DEVICE_CREATE_MULTI_DEVICE_IQ_MATCH_BIT"/>
</enums>
```

Several other attributes of the `<enums>` tag can be set. In this case, the `type` attribute is set to `"bitmask"`, indicating that the individual enumerants represent elements of a bitmask.

The individual <enum> tags define the enumerants, just like the definition for compile-time constants described above. In this case, because the enumerants are bitfields, their values are specified using the bitpos attribute. The value of this attribute must be an integer in the range $[0, 31]$ specifying a single bit number, and the resulting value is printed as a hexadecimal constant corresponding to that bit.

It is also possible to specify enumerant values using the value attribute, in which case the specified numeric value is passed through to the C header unchanged.

A.6 How to Add A Command

For this example, assume we want to define the command:

```
VKAPI_ATTR VkResult VKAPI_CALL vkCreateInstance(
    const VkInstanceCreateInfo*                 pCreateInfo,
    VkInstance*                                 pInstance);
```

Commands must always be explicitly required in the <feature> tag. In that tag, you can use an existing <require> block including API features which the new command should be grouped with, or define a new block. For this example, add a new block, and require the command by using the <command> tag inside that block:

```
<!-- SECTION: Vulkan API interface definitions -->
<feature api="vulkan" name="VK_VERSION_1_0" number="1.0">
    ...
    <require comment="Device initialization">
        <command name="vkCreateInstance"/>
    </require>
    ...
</feature>
```

The <require> block may include a comment attribute whose value is a descriptive comment of the contents required within that block. The comment is not currently used in header generation, but might be in the future, so use comments which are polite and meaningful to users of vulkan.h.

Then go to the <commands> tag and add a new <command> tag defining the command, preferably sorted into alphabetic order with other commands for ease of reading, as follows:

```
<!-- SECTION: Vulkan command definitions -->
<commands>
    ...
    <command>
        <proto><type>VkResult</type>
               <name>vkCreateInstance</name></proto>
        <param>const <type>VkInstanceCreateInfo</type>*
               <name>pCreateInfo</name></param>
        <param><type>VkInstance</type>*
```

```
                <name>pInstance</name></param>
    </command>
    ...
</commands>
```

The <proto> tag defines the return type and function name of the command. The <param> tags define the command's parameters in the order in which they're passed, including the parameter type and name. The contents are laid out in the same way as the structure <member> tags described previously.

A.7 More Complicated API Representations

The registry schema can represent a good deal of additional information, for example by creating multiple <feature> tags defining different API versions and extensions. This capability is not yet relevant to Vulkan. Those capabilities will be documented as they are needed.

A.8 More Complicated Output Formats And Other Languages

The registry schema is oriented towards C-language APIs. Types and commands are defined using syntax which is a subset of C, especially for structure members and command parameters. It would be possible to use a language-independent syntax for representing such information, but since we are writing a C API, any such representation would have to be converted into C anyway at some stage.

The vulkan.h header is written using an **output generator** object in the Python scripts. This output generator is specialized for C, but the design of the scripts is intended to support writing output generators for other languages as well as purposes such as documentation (e.g. generating Asciidoc fragments corresponding to types and commands for use in the API specification and reference pages). When targeting other languages, the amount of parsing required to convert type declarations into other languages is small. However, it will probably be necessary to modify some of the boilerplate C text, or specialize the tags by language, to support such generators.

A.9 Additional Semantic Tagging

The schema is being extended to support semantic tags describing various properties of API features, such as:

- constraints on allowed scalar values to function parameters (non-NULL, normalized floating-point, etc.)
- length of arrays corresponding to function pointer parameters
- miscellaneous properties of commands such as whether the application or system is responsible for threadsafe use; which queues they may be issued on; whether they are aliases or otherwise related to other commands; etc.

These tags will be used by other tools for purposes such as helping create validation layers, generating serialization code, and so on. We'd like to eventually represent everything about the API that is amenable to automatic processing within the registry schema. Please make suggestions on the Gitlab issue tracker.

A.10 Stability of the XML Database and Schema

The Vulkan XML schema is evolving in response to corresponding changes in the Vulkan API and ecosystem. Most such change will probably be confined to adding attributes to existing tags and properly expressing the relationships to them, and making API changes corresponding to accepted feature requests. Changes to the schema should be described in the change log of this document (see section B). Changes to the `.xml` files and Python scripts are logged in Gitlab history.

B Change Log

- 2016/02/22 - Change math markup in `len` attributes to use asciidoc "latexmath:[and]" delimiters.
- 2016/02/19 - Add `successcodes` and `errorcodes` attributes of `<command>` tags. Add a subsection to the introduction describing the schema choices and how to file issues against the registry.
- 2016/02/07 - Add `vendorids` tags for Khronos vendor IDs.
- 2015/12/10 - Add `author` and `contact` attributes for `<extension>` tags.
- 2015/12/07 - Move `vulkan/vulkan.h` to a subdirectory.
- 2015/12/01 - Add `<tags>` tags for author tags.
- 2015/11/18 - Bring documentation and schema up to date for extension enumerants.
- 2015/11/02 - Bring documentation and schema up to date with several recent merges, including `<validity>` tags. Still out of date WRT extension enumerants, but that will change soon.
- 2015/09/08 - Rename `threadsafe` attribute to `externsync`, and `implicitunsafeparams` tag to `implicitexternsync`.
- 2015/09/07 - Update `<command>` tag description to remove the `threadsafe` attribute and replace it with a combination of `threadunsafe` attributes on individual parameters, and `<implicitunsafeparams>` tags describing additional unsafe objects for the command.
- 2015/08/04 - Add `basetype` and `funcpointer` `category` values for type tags, and explain the intended use and order in which types in each category are emitted.

- 2015/07/02 - Update description of Makefile targets. Add descriptions of `threadsafe`, `queues`, and `renderpass` attributes of `<command>` tags in section 12, and of `modified` attributes of `<param>` tags in section 12.4
- 2015/06/17 - add descriptions of allowed `category` attribute values of `<type>` tags, used to group and sort related categories of declarations together in the generated header.
- 2015/06/04 - Add examples of making changes and additions to the registry in appendix A.
- 2015/06/03 - Move location to new "vulkan" Git repository. Add definition of `<type>` tags for C struct/unions. Start adding examples of making changes in appendix A.
- 2015/06/02 - Branch from OpenGL specfile documentation and bring up to date with current Vulkan schema.