

# Asciidoctor Workflow

Presented to the ANARI Working Group

Jon Leech 2020-03-18

(last updated 2020-06-15)

# What is AsciiDoctor?

Text-based lightweight markup language, similar to Markdown (and many others). Significant user base. Khronos contributes to the AsciiDoctor project (<https://asciidoc.org>)

Implementations in Ruby (desktop / command-line), JavaScript (AsciiDoctor.js), Java (AsciiDoctorJ).

Supports tables, PNG / SVG images, other usual stuff.

Unextended asciidoc markup can be rendered directly in a browser, using the AsciiDoctor.js preview extension. It is also rendered directly by [GitHub](#) and gitlab servers (can completely replace Markdown/.md files).

There are issues with complex documents, rendering mathematics, etc. in unextended asciidoc, but it may be suitable for your project, and is simplest.

# Khronos use of AsciiDoctor

Started with Vulkan. Now also used for SPIR-V, OpenXR, OpenCL, OpenVX, and Data Format specs. Several others are transitioning. Lots of collective experience.

Vulkan and OpenXR are the most aggressive users. We do *not* use unextended asciidoctor, so must use a build step to generate output HTML / PDF documents from source when doing “releases”, a.k.a. minor spec updates.

Workflow is like a software project in gitlab. Master branch contains the current spec, merge requests against master to fix / add features/extensions, WG review and signoff, periodic releases.

# Public Collaboration

We keep the public specification on github and accept pull requests there for fixes / contributions from developers outside Khronos, but most development work, including pre-release spec updates and Khronos / vendor extensions, happens in gitlab. GitHub / gitlab repos are synced when doing releases.

Our experience has been a relatively small number of public contributors, but they suggest useful things, find errors, etc.

(Then there's always that one gadfly who wants to completely change how you're doing everything because it's WRONG™).

How other WGs do public collaboration is a downstream decision for you at this early stage, no impact on spec markup / toolchain. But it would be a good idea to follow processes, licensing decisions, etc. used by most other Khronos APIs. Neil Trevett, Brad Biddle, Jon Leech are all resources for this.

# Asciidoctor as part of Vulkan/OpenXR toolchain

We use a complex spec development toolchain intended for a C API. Both Vulkan and OpenXR generate C++ wrappers as well, using different techniques. The other APIs mostly use subsets of these tools due to different needs / scope. SPIR-V uses asciidoctor and some tooling specific to their spec.

You can write a spec in asciidoctor markup without doing **any** of what I'm about to describe! And this toolchain has a steeper learning curve than just writing asciidoctor. But we've found this toolchain / ecosystem to have a lot of value, so I'm describing it to see if it is may be appropriate for your needs.

## Toolchain components

- XML description of the API (realistically, only C - C++ support would be major project)
- Generator scripts which operate on the XML to create transient and publishable artifacts
- Extended asciidoctor specifications which incorporate some of the transient artifacts
- Downstream API ecosystem uses our XML for many other purposes

# XML API description

XML tags for describing functions, structures, enumerated types, other C data types. The schema includes bits of C at the lowest level with imbedded semantic tags, for example

```
<type category="struct" name="VkBaseInStructure">  
  <member><type>VkStructureType</type> <name>sType</name></member>  
  <member>const struct <type>VkBaseInStructure</type>* <name>pNext</name></member>  
</type>
```

->

```
typedef struct VkBaseInStructure {  
    VkStructureType sType;  
    const struct VkBaseInStructure *pNext;  
};
```

# Generator Scripts

Python scripts read the XML and a set of required versions / extensions, build up internal data structures, and use a plugin scheme to call “generators” which transform the selected API interfaces into artifacts such as (incomplete list):

- C header files
- C code fragments defining APIs (prototypes, type definitions, etc.) for use in the specification
  - Eliminates inconsistency between spec and headers that can happen when both are hand-written
- Metainformation about extensions and versions for incorporation in appendices of the spec
- Automatically generated validity statements for incorporation in the spec (essentially, parameter validation based on types, and sometimes additional XML attributes describing a function parameter / structure member)

# XML schema

All API entities / formal names are tagged. Different feature sets of the API (core versions, extensions) are defined by reference to the API entities they incorporate.

XML is also a registry for shared namespaces (global Vk\*, enum values in enumerated types, etc.)

Lots of people on GitHub have strong ideas about how the schema should differ, but they haven't presented a compelling reason for us to change (also, they aren't a unified block). Significant schema changes would require a lot of additional toolchain work.

# Downstream Generator Scripts

Many downstream users of the XML

- [Intercept layers](#) sitting above the drivers / core API
  - Loader - provides a cross-vendor method to load drivers and dispatch at runtime
  - Validation layers - generate validation code, similar to validity statements
  - Debugging layers - w/higher-level validation (object lifetimes, multi-threading usage)
  - Generator scripts are a small but important part of each of these layers
- C++ Binding Generators
  - Separate project for Vulkan (Vulkan-Hpp), OpenXR has a different approach
- Other language bindings
  - Outside developers generate language bindings based on XML (quite a few of these). Some of them are not completely happy with the XML schema but still make effective use of it.
- No requirement to use our scripts - some people parse XML directly
  - But the scripts make it easier for most purposes

# Specification & Toolchain

We use asciidoctor extensions (Ruby plugins) for math (using LaTeX markup), PDF generation, extraction of some types of formal language (hand-written validity statements) from the document, “chunked” HTML generation. Getting a build environment on Linux or Windows WSL (some MacOS users too) requires a lot of dependencies - we now supply a Docker Linux image with all the spec tooling pre-installed.

Strong constraints on markup (in the Vulkan “Style Guide”) include semantic markup (custom macros for tagging API entities), enable automatic reference page extraction, other consistency guidelines. Vulkan reppages are entirely generated from the spec. For some idea of what the markup looks like, refer to

<https://github.com/KhronosGroup/Vulkan-Docs/blob/master/chapters/initialization.txt>

Ryan Pavlik (OpenXR spec editor) wrote a very nice set of markup-checking scripts which catch many common issues, incorporated as part of spec repo CI.

# Why all this mechanism?

Vulkan started off as a relatively small API, but we expected it to live a long time and be extended a lot, and it has (4+ years old, hundreds of vendor and Khronos extensions). Spec document and generated specs have become huge - full spec including all extensions more than 1,000 PDF pages!

Keeping the spec internally consistent and the toolchain working well has been a priority. Toolchain has had improvements and bugfixes over time but has withstood this growth well.

Spec authors need some assistance at first following the style guide, but checker script and assistance from other authors / spec editor helps. Dozens of people have written substantial contributions to the Vulkan spec at this point.

# Next Steps

ANARI could just use unextended asciidoctor markup. Then you're basically on your own, but writing spec markup is simpler (at least at first). General process / licensing guidelines for working on GitHub would still be relevant in this case.

If you want to incorporate some or all of the Vulkan / OpenXR toolchain, I am available to help<sup>1</sup>. Others might be able to help as well, though they all work for member companies so are time-constrained. I'd want to work with your spec editor(s) to setup a framework for the spec and toolchain, give enough examples to help you start writing, and then back off and answer questions / fix scripting issues as needed until you are comfortable with it.

<sup>1</sup>I'm a contractor paid by Khronos, but most of my time is spent on Vulkan - can discuss w/Neil & Emily.