Vulkan Unified Validation Layer

Mark Lobodzinski, LunarG
April 2019

Introduction

Early in the development of the Vulkan validation layers, there was A Layer For Everything -- nearly a dozen individual validation layers. With the introduction of handle wrapping and the increased specialization of types of layer checks, the order in which the layers were loaded became critical for correct operation. However, communicating the optimal layer ordering, if not outright enforcing it, became increasingly difficult.

The issue was ameliorated on desktop platforms by the introduction of a meta-layer that abstracted the physical layer set, but this was only a partial solution and was unavailable on mobile platforms. Over time, the number of discrete validation layers were reduced to five but the goal for usability, maintainability, and messaging remained that of a single, unified validation layer.

A Need For Change

Over time, the number and names of the canonical validation layers have changed several times, and, in fact, this was one of the issues that the move towards a single layer was going to solve. However, there were additional considerations driving this effort.

Issues with Multiple Layers

Layer consolidation had the goal of solving three main problems: Cross-platform messaging and configuration, issues with validation layer load order, and maintenance and usability.

Layer Order

To function most effectively, validation layers must be loaded in a particular order. For instance, parameter validation and object lifetime checking should come early in the stack, reducing or even eliminating the need for other layers to duplicate null-pointer or invalid object checks. Handle wrapping must take place after all validation is complete, just before the display driver. With multiple layers, communicating the optimal layer order to users and developers became

quite significant both in effort and importance. Significant layer-development time was spent tracking down developer issues ultimately attributable to loading incomplete or disordered layer sets, resulting in invalid results or crashes. Also, specifying the order through the command-line or an environment variable became quite cumbersome.

How to specify optimal layer order

```
VK_INSTANCE_LAYERS=VK_LAYER_GOOGLE_threading;VK_LAYER_LUNARG_paramet
er_validation;VK_LAYER_LUNARG_object_tracker;VK_LAYER_LUNARG_core_validation;
VK_LAYER_GOOGLE_unique_objects;
```

As a temporary work-around, a ‘meta-layer’ was introduced to alleviate the issues. This was a layer identifier with a recognized name (`VK_LAYER_LUNARG_standard_validation`) that was intercepted by the loader that then loaded the appropriate layer set in the optimal order. Significantly, this solution only worked for desktop platforms, leaving mobile platforms at a disadvantage and increasing confusion as to how this pseudo-layer related to other validation layers. This became a difficult communication issue in itself.

Platform Differences and Messaging

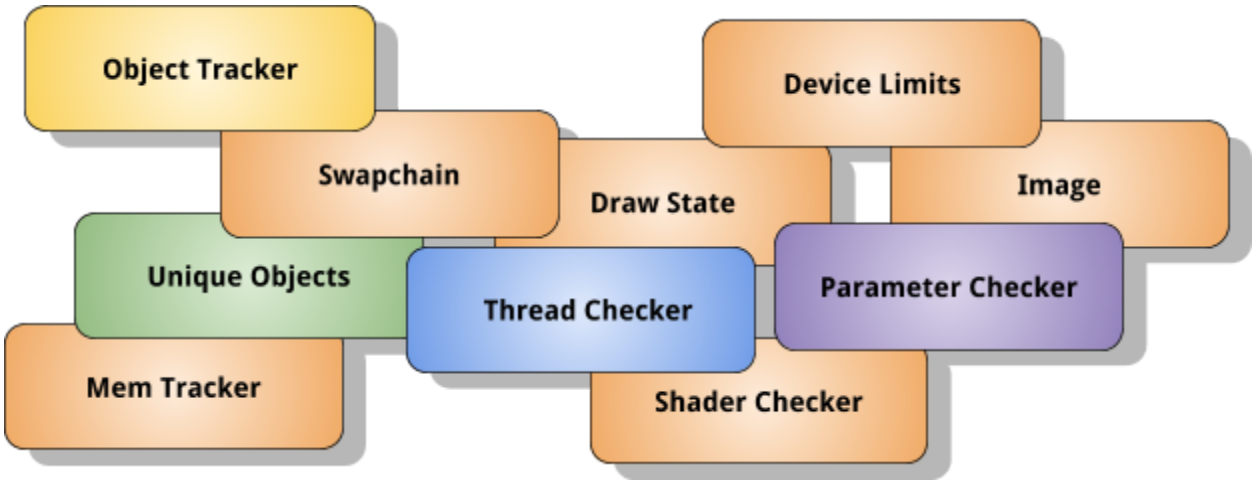
Android implementations are less flexible when it comes to changes in the canonical set of validation layers. For instance, Android platforms are infrequently updated compared to repository updates or even SDK releases. It is important that these platforms have a stable set of layers. A single layer fulfills this requirement with the advantage that layer order issues then become moot.

Additionally, due to the Vulkan layer naming scheme, the collection of validation layers had multiple company names embedded in them (*i.e.*, Google and LunarG), which might lead to confusion for developers expecting Vulkan validation layers. Moving to a single layer also presented the opportunity of making the layer naming more clear and understandable.

Maintainability and Ease Of Use

As mentioned previously, the initial Vulkan bring-up effort saw a proliferation of distinct validation layers.

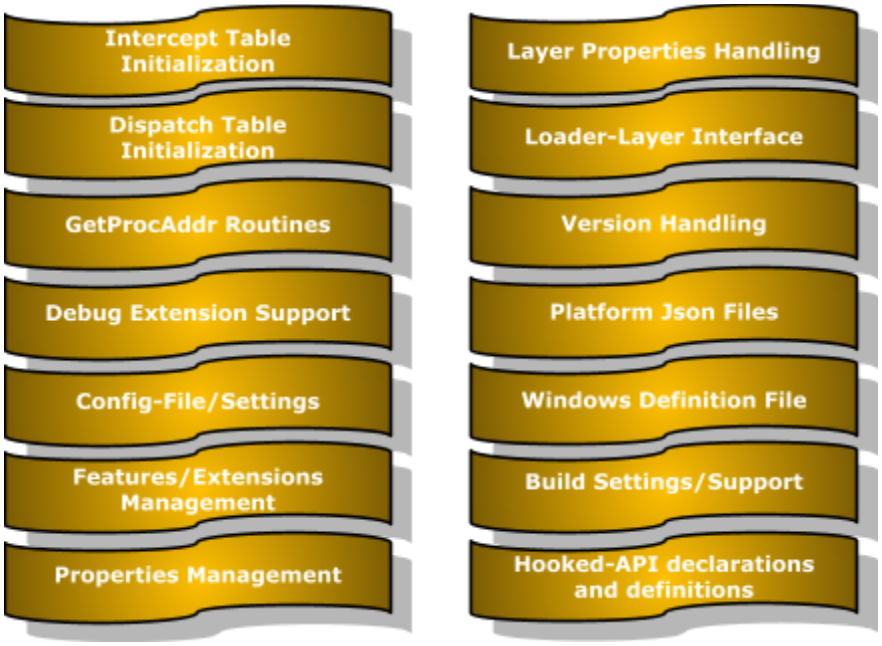
Historical Validation Layers



At that time, this seemed reasonable as layer content was in the early stages of development and individual layers eased parallel development across a large number of people. This also took great advantage of Vulkan's novel layer architecture. As development progressed into the long-term, disadvantages stemming from the profusion of layers became apparent and in some cases began to seriously hinder development.

Each of the validation layers, as well as other utility layers, required its own *complete* set of layer infrastructure, consisting of many supporting elements.

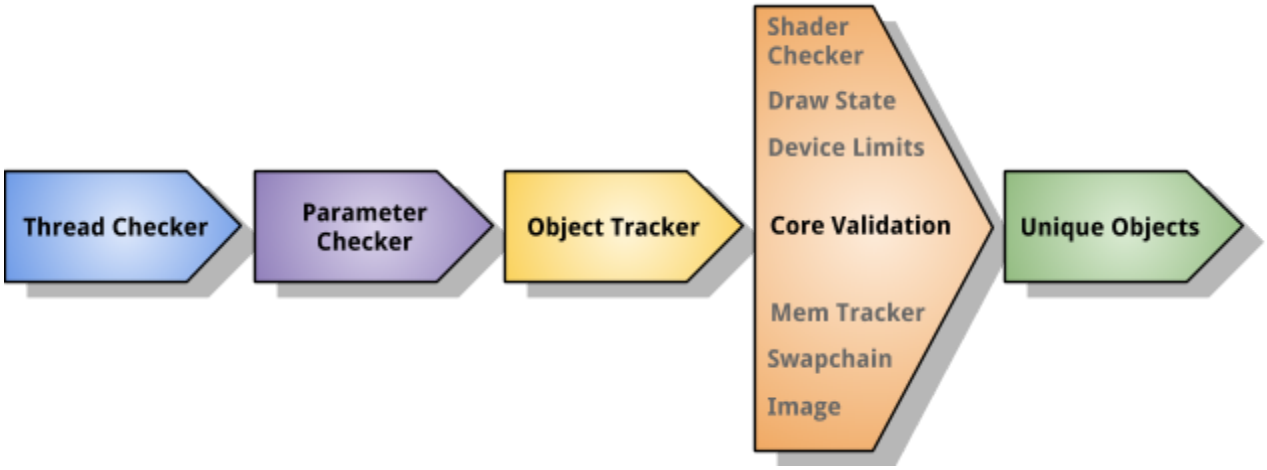
Layer Infrastructure Elements



Some progress was made towards sharing layer infrastructure (utils, logging, config), but this did little to prevent layers from diverging greatly in their particular implementations. Making cross-layer changes became almost prohibitively expensive in time and manpower.

In order to reduce duplication of state information and to minimize the frustrations of maintaining so many layers, the number of discrete layers was gradually reduced, mostly through brute-force coalescing of source code. These efforts resulted in the introduction of the **core validation layer**, which eventually subsumed half-a-dozen existing layers.

Canonical Validation Layers in Optimal Order



Code Separation and Organization

As the number of validation layers shrank, their complexity grew. Lacking a consistent layer architecture, layers used different conventions and organizations to implement layer support functions and validation source code, and it was common that the two were commingled. A poorly defined separation between layer management source code and validation source code made it more difficult for new contributors.

Code Generation and API Updates

Several of the canonical layers used Vulkan registry-based code generation for their source code. Some layers were completely code generated, some partially so, while core validation uses no generation.

Code generation of layer infrastructure source code can be extremely useful, as new APIs, extensions, and changes are picked up without user intervention, and it becomes unnecessary for contributors to add infrastructure support for new Vulkan features. This keeps the layers up-to-date and accurate to the specification, thereby reducing code-divergence and support issues.

The most recent layer consolidation process resulted in a nearly 5,000-line reduction (~10%) in layer infrastructure source code.

Layer Consolidation

To address the issues of layer order, naming, messaging, and ease-of-use and maintainability, the determination was made to combine the canonical set of validation layers into a single validation layer. Moving to a single layer would immediately equalize the major platform disparity issues while also instantly resolving layer-order issues.

A single layer also means that five divergent and independent sets of layer infrastructure could be replaced by a single well-maintained set.

In developing a plan for the layer consolidation, the following constraints were identified:

- Zero-downtime for the validation codebase
- Implementation should occur in small, well-defined stages
- Development should occur within the existing source base
- Validation source code should be separated from layer infrastructure source code
- Canonical layers (development, build, and execution) must coexist seamlessly with the consolidated layer at all times
- Validation source code must be shared between the two layer sets

A small handful of implementation alternatives were considered. The first was to continue the previous approach of coalescing layers, copying the related source code from each layer into the combined layer, and then combining each layer's data into a single, monolithic layer data structure.

Brute-force Consolidation Approach

```
khronos_layer_ApiCall(parameters) {

    Source copied from thread checker layer;
    Source copied from Parameter validation layer;
    Source copied from Object tracker layer;
    Source copied from Core validation layer;

    // Down-chain API call itself
    vkAPICall(parameters);

    Maybe more thread checker source;
    Maybe more parameter validation source;
    Maybe more object tracker source;
    Maybe more core validation source;
}
```

This problematic approach would result in excessive disruption to the codebase and significant impacts to quality and reliability would be impossible to avoid.

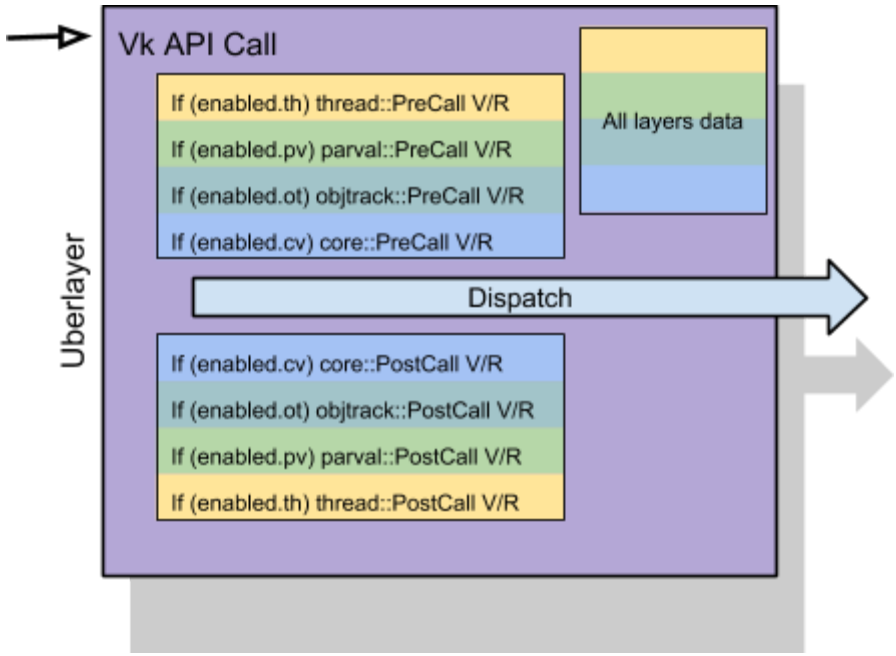
A better option would be to extend the (longstanding, but only partially implemented) standard layer architecture to all layers, potentially easing the consolidation effort. This previously-developed design organized validation source code into subroutines containing the before-call validation code, before-call state update code, and after-call state update code.

Standard Layer Architecture

```
layer_ApiCall(parameters) {

    // All validation-related source
    PreCallValidate(parameters);

    // Any before-APICall state updates
    PreCallRecord(parameters);

    // Down-chain API call itself
    vkAPICall(parameters);

    // Any after-API call state updates
    PostCallRecord(parameters);
}
```

If this architecture was extended to all layers, collecting the Validate- and Record- routines into a single layer and calling them consecutively would facilitate consolidation.

Consolidation using Standard Layer Architecture

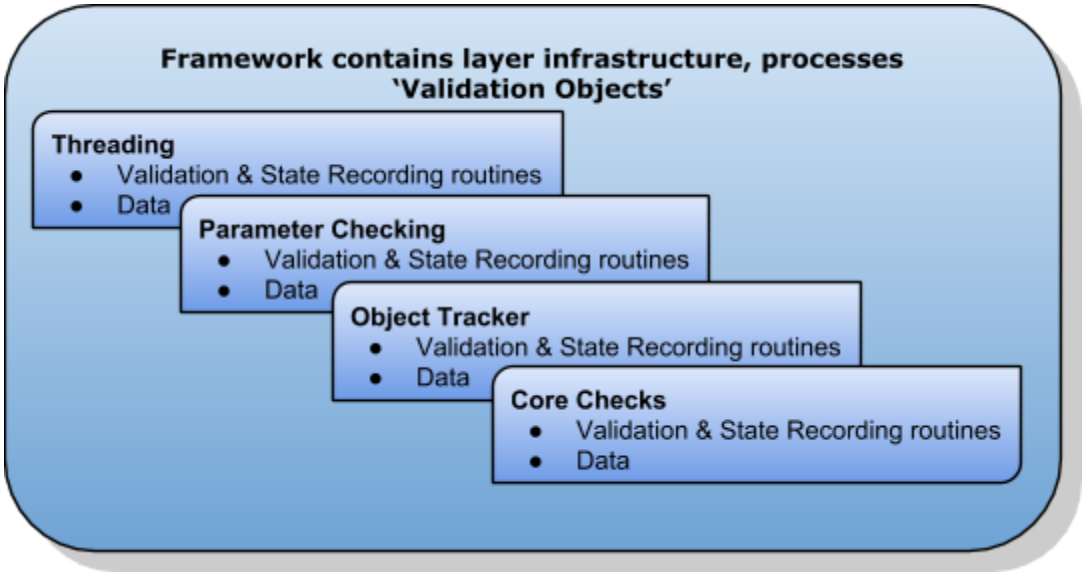


However, the difficult problem of merging the layer data from the separate layer implementations would remain.

Final Approach

The chosen approach was to rely on the second option and extend it using a heavily modified layer framework based on the Vulkan Layer Factory. In this architecture, the data and pre/post calls for each layer would live in a *validation object*. Polymorphism allowed methods in each layer's object to be called independently, and all layer data would be encapsulated in the object itself. This option required some additional up-front work, but this was mostly offset by the simpler nature of the changes, which were low-impact and could be done in small stages.

The Framework (chassis) Processes *Validation Objects*



Ultimately, each existing layer was migrated to use this new framework, which became the 'layer chassis.' After the five existing layers had moved to the new validation object architecture, combining them into a single layer was a straightforward task.

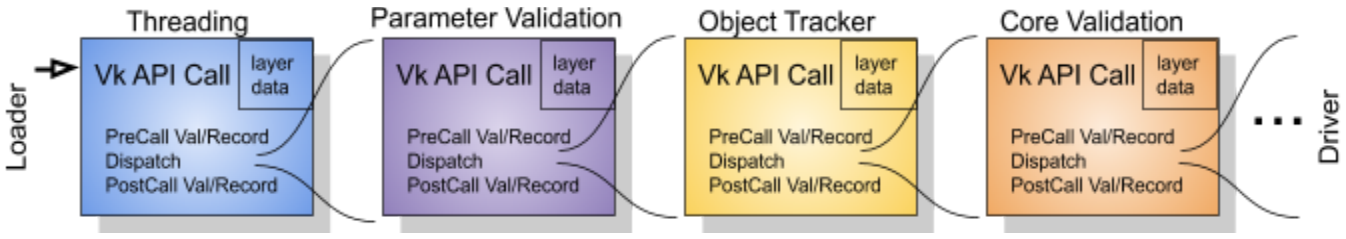
The Layer Chassis

The layer chassis implements a framework to process *validation objects*, handling many common functions and all dispatch control.

Vulkan Layer Dispatching

The Vulkan layer call structure is implemented such that, for a given API, each layer in the chain is called sequentially. Layers not hooking a specific API call are skipped, and if no layer hooks an API-call, the chain may contain only the display driver.

The Standard Validation Call Chain

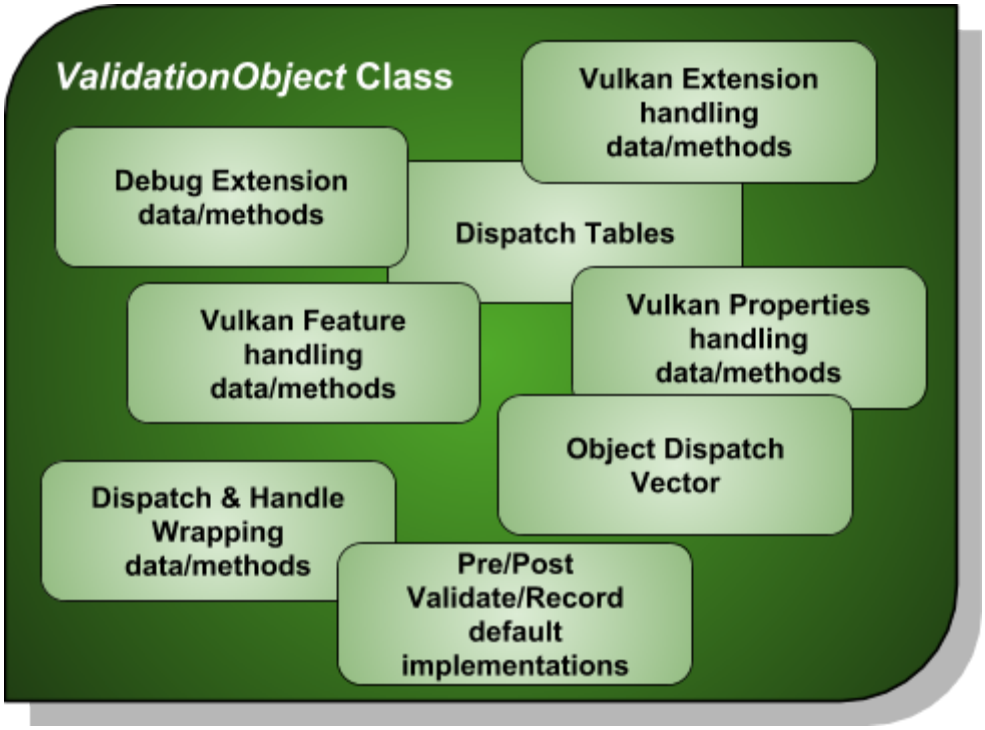


In each layer, persistent state was separated into global per-instance and per-device data structures and kept in a map. Each instance (and all physical devices for that instance) mapped back to a single *instance layer data* structure. Likewise, all devices (and related command buffers and queues) mapped back to a single owning *device layer data* structure

The Validation Object Architecture

The new validation object architecture would need to maintain this separation of data, as well as provide a form of dispatching that would normally be done by the layer chain. Thus, a new base *validation object* class was defined. This class contained much of the data held in the instance layer-data maps of the existing layers.

ValidationObject Class Data



Each of the old layers has been refactored as a *validation object*, overriding default pre/post API functions where needed and extending the class with layer-specific data.

ValidationObject Child Objects

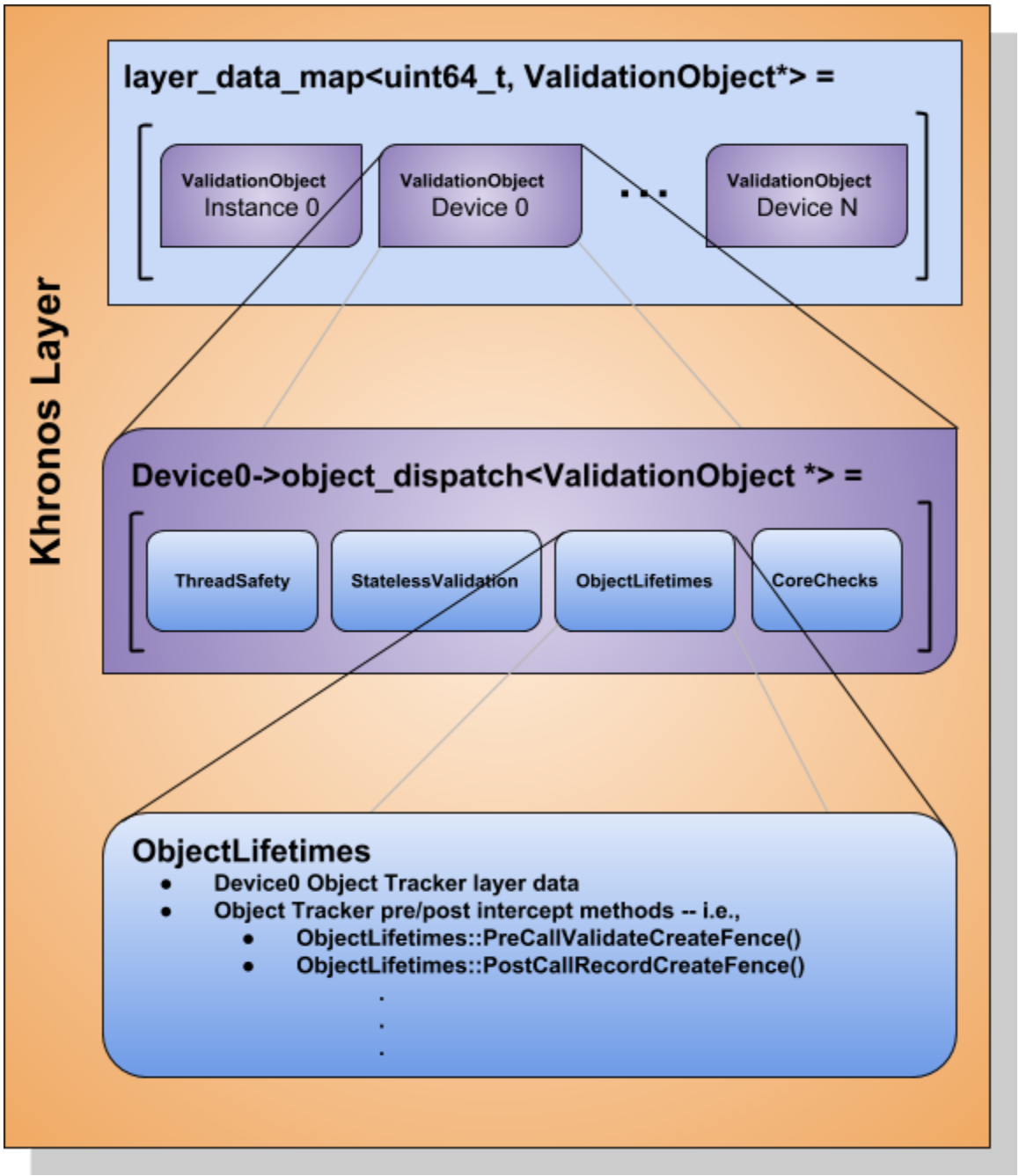
Old Layer Name	`ValidationObject` Child Class
threading	**`ThreadSafety`**
parameter_validation	**`StatelessValidation`**
object_tracker	**`ObjectLifetimes`**
core_validation	**`CoreChecks`**
unique_objects	*Implemented in base class*

Note that the handle-wrapping carried out by the `unique_objects` layer required some additional special consideration. This utility layer is responsible for ensuring that each handle the validation layers encounter is unique, which improves validation in the cases of duplicate or reused handles from the driver. The special consideration was that validation layers, in the course of normal operation, sometimes *originate* API calls to the driver. To guarantee that in these cases the handle wrapping was also performed, the `unique_objects` functionality was implemented as part of the layer chassis. Each API-call has a dispatch function, which (optionally) wraps/unwraps Vulkan handles and calls down the chain, and these dispatch functions can be used by the chassis dispatch calls as well as by other *validation objects*.

Layer Data Organization

Each Vulkan instance and device has its own instances of each of these child objects held in the *Object Dispatch Vector* of the instance/device *validation objects*. The `layer_data` map now holds pointers to a *validation object* created for each instance and each device.

Layer Chassis Data Layout

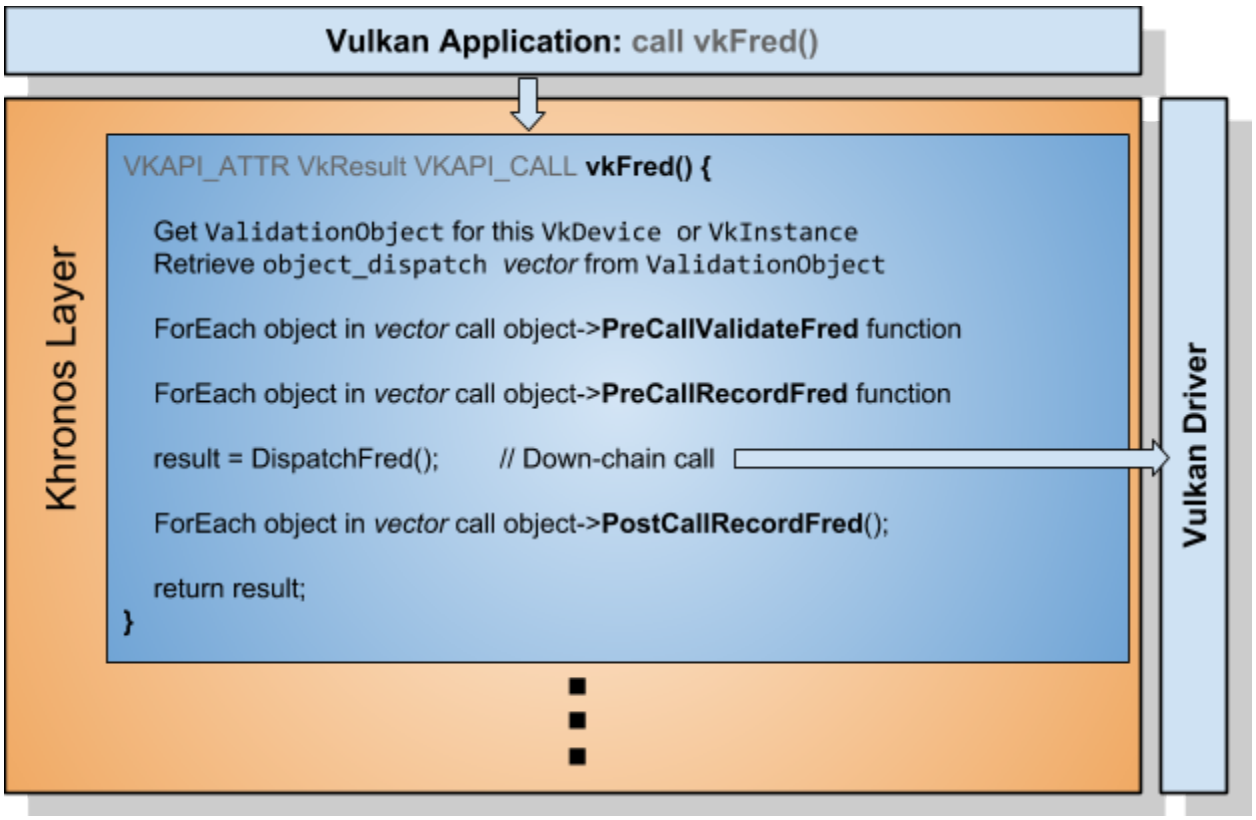


Layer Chassis Dispatching

The base class *ValidationObject* has default methods for all pre/post intercept functions, and each child *validation object* (*e.g.*, ThreadSafety, StatelessValidation, etc.) can override any of the functions that they need to intercept. For a given API, the chassis will call the virtual

functions in each child class for each of the three intercept points: `PreCallValidate`, `PreCallRecord`, or `PostCallRecord`.

Sample Chassis API Call Control Dispatching



Pre/Post Call Function Signatures

Other than a handful of special cases, function signatures for the `PreCallValidate` and `PreCallRecord` methods match the API call signature exactly. However, `PostCallRecord` methods do not follow this pattern. In order to allow these functions the necessary context, down-chain call return values are passed into these functions as the final parameter. For instance, the standard Vulkan signature for vkQueuePresentKHR is:

```
VkResult vkQueuePresentKHR(VkQueue, const VkPresentInfoKHR *) {};
```

But the `PostCallRecord` method for this function is modified to take the `VkResult` parameter:

```
void PostCallRecordQueuePresentKHR(VkQueue, const VkPresentInfoKHR *, VkResult) {};
```

Standard Dispatching Scheme Special Cases

There is a small number of cases where the standard Pre/Post architecture has not been followed, resulting in changes to the standard chassis function signatures for these APIs. At this time, all of these cases are in the `CoreChecks` *ValidationObject*. These changes fall into performance-related persistent state cases, or changes to support GPU-assisted validation.

Each of these calls is special-cased in the layer chassis, and for each, an additional *validation object* default implementation is supplied with the custom function signature that then calls back into the original method if not overridden. For example, to allow *ValidationObjects* other than `CoreChecks` to deal with the custom saved-state parameter (**`ads_state`**) used by `vkAllocateDescriptorSets()`, the default implementation falls back to the standard default implementation.

Custom Function Default Fallback

```
virtual bool ValidationObject::PreCallValidateAllocateDescriptorSets(VkDevice,
    const VkDescriptorSetAllocateInfo*, VkDescriptorSet*, void* ads_state)  {
    return PreCallValidateAllocateDescriptorSets(device, pAllocateInfo, pDescriptorSets);
};
```

The typical situation for the performance cases is that complex data structures must be created and populated for the validation *and* recording steps, and that the potential performance impact of repeating this data consolidation step was prohibitive. The following APIs save some state locally for performance benefit:

- `vkCreateGraphicsPipelines()`
- `vkCreateComputePipelines()`
- `vkCreateRayTracingPipelinesNV()`

GPU-assisted validation modifies the down-chain call parameters for some calls. These are:

- `vkCreateDevice()`
- `vkCreateGraphicsPipelines()`
- `vkCreatePipelineLayout()`
- `vkCreateShaderModule()`
- `vkAllocateDescriptorSets()`

Validation Thread Safety

To prevent race conditions and maintain the integrity of validation state data, validation and recording routines are typically protected by locks. In the legacy architecture, each layer relied on a single global lock. In the validation object architecture each *ValidationObject* has its own

lock. Before dispatching pre- or post-API calls to child *ValidationObjects*, the lock for that particular child object is obtained using a virtual function. Depending on its own needs, the child object determines if the function actually implements a lock or not. For instance, `ThreadSafety` requires that no locks are held during its validation and recording steps.

Example Layer Chassis Vulkan API Call Showing Locking

```
VKAPI_ATTR VkResult VKAPI_CALL CreateEvent(VkDevice device, const VkEventCreateInfo* pCreateInfo,
                                           const VkAllocationCallbacks* pAllocator, VkEvent* pEvent) {
    auto layer_data = GetLayerDataPtr(get_dispatch_key(device), layer_data_map);
    bool skip = false;
    for (auto intercept : layer_data->object_dispatch) {
        auto lock = intercept->write_lock();
        skip |= intercept->PreCallValidateCreateEvent(device, pCreateInfo, pAllocator, pEvent);
        if (skip) return VK_ERROR_VALIDATION_FAILED_EXT;
    }
    for (auto intercept : layer_data->object_dispatch) {
        auto lock = intercept->write_lock();
        intercept->PreCallRecordCreateEvent(device, pCreateInfo, pAllocator, pEvent);
    }
    VkResult result = DispatchCreateEvent(device, pCreateInfo, pAllocator, pEvent);
    for (auto intercept : layer_data->object_dispatch) {
        auto lock = intercept->write_lock();
        intercept->PostCallRecordCreateEvent(device, pCreateInfo, pAllocator, pEvent, result);
    }
    return result;
}
```

Feature Control

Disabling a layer in the multi-layer model was straightforward, if not convenient -- simply avoid loading the layer. However, achieving the same level of control in the validation object architecture required the provision of other control methods, The layer chassis supports three methods; in order of usefulness, these are the `VK_EXT_validation_features` extension, the `vk_layer_settings.txt` file, and environment variables.

Validation Features Extension

Applications can control the enabling or disabling of *validation features* through the `VK_EXT_validation_features` extension. Though it offers other options, for the purposes of this paper, the focus is placed on the layer-control aspects of this extension.

By specifying one or more of the following enum values at application CreateInstance-time, features corresponding to each of the legacy layers can be disabled.

Mapping of Layer Names to Corresponding *Validation Features*

Legacy Layer Name	Validation Features Disable Enum Value
Threading	VK_VALIDATION_FEATURE_DISABLE_THREAD_SAFETY_EXT
Parameter Validation	VK_VALIDATION_FEATURE_DISABLE_API_PARAMETERS_EXT
Object Tracker	VK_VALIDATION_FEATURE_DISABLE_OBJECT_LIFETIMES_EXT
Core Validation	VK_VALIDATION_FEATURE_DISABLE_CORE_CHECKS_EXT
Unique Objects	VK_VALIDATION_FEATURE_DISABLE_UNIQUE_HANDLES_EXT

Configuration File

Features can also be controlled through the standard `vk_layer_settings.txt` file. The configuration entry used for this purpose is `'disables.'`

Sample Settings File Disable Entry

```
khronos_validation.disables = VK_VALIDATION_FEATURE_DISABLE_THREAD_SAFETY_EXT
```

Adding this entry in the layer config file will result in the `Thread Safety` feature being disabled during the validation of the application.

Environment Variables

Similarly, an environment variable can be used to produce the same result.

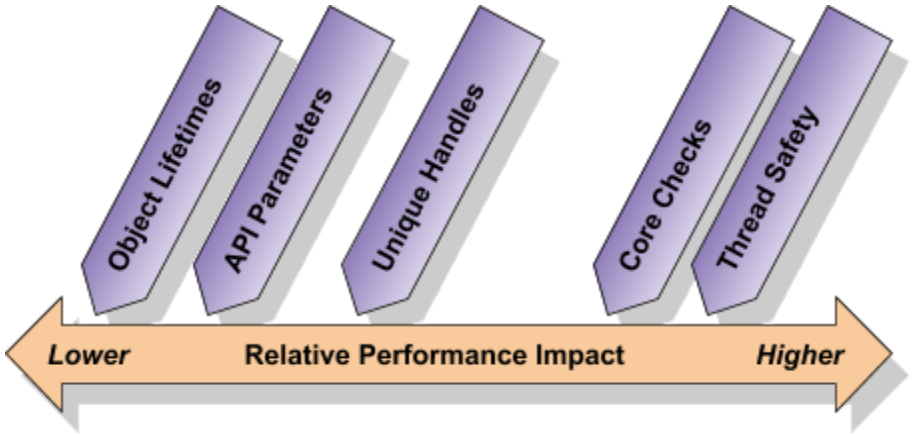
Sample Environment Variable Disable Entry

```
Set VK_LAYER_DISABLES=VK_VALIDATION_FEATURE_DISABLE_THREAD_SAFETY_EXT
```

Validation Workflow

Each *validation feature* is accompanied by a performance cost. For some projects, it may be that running with a subset of features is a more optimal workflow.

Relative Performance Impact of Validation Features



Running an application with only `Object Lifetimes` and `API Parameters` enabled would have the least impact on performance, yet would catch most of the obvious errors that might later cause crashes. `CoreChecks` might be enabled after the application is running cleanly with the first two features. `Unique Handles` should be enabled for hard-to-track-down object lifetime or core checks issues, while perhaps the `Thread Safety` feature is only occasionally enabled.

Transition Away from Legacy Layers

The Vulkan-ValidationLayers (and supporting repositories) and the Vulkan SDK as of release 1.1.106 will use the unified Khronos validation layer as the default validation layer in all cases. However, the five legacy layers still exist, are present in the repository and the SDK, and will function exactly as before for a period of time, perhaps several weeks or months. After this time, the legacy layers and references to them will be removed from the repository, significantly speeding up build-times and reducing complexity in the chassis. The changes will be propagated through the following Vulkan SDK, and all documentation will be updated accordingly.

Summary

As time has provided the perspective of distance to the view of validation layer development, so has experience led to the realization that having more validation layers is not necessarily better. In fact, having a single layer will go a long way towards solving some of the problems responsible for that hard-won experience: issues with layer ordering, platform disparities, and long- and short-term maintenance and updates.

Ultimately, the decision was made to consolidate the remaining five validation layers and the necessary resources were allocated, but some important constraints were included. The consolidation had to be implemented in relatively small stages that would minimally affect other development. At all times, the current working set of layers must coexist with and work alongside the new consolidated layer. Most importantly, the old and new layers must share all validation source code.

These requirements narrowed the set of possible solutions, leading to a 'validation object' architecture. In this new design, each individual layer becomes a *validation object*, containing all of the layer's validation routines and data. *Validation objects* are then processed by a newly-developed validation framework known as the 'layer chassis,' developed with experience gained from the Vulkan Layer Factory.

Each existing layer was incrementally converted to this *validation object*-based architecture, in most cases transparently to other ongoing development. Upon completion of this phase, the legacy layers existed as five individual instantiations of the framework, each having a single *validation object*. The new, consolidated layer quickly followed, implemented as a single instantiation of the framework but processing multiple *validation objects*. This consolidated layer is named the Khronos validation layer -- `VK_LAYER_KHRONOS_validation`.

The new architecture allows much simpler, straightforward messaging concerning which layers should be loaded, and in that order, bringing parity across platforms. A reduction in the amount of source code, a more logical source code layout, and a more robust layer infrastructure are significant steps along the way to a healthier Vulkan ecosystem.

Future

Layer chassis work is not complete. Here are a few items under consideration for the current implementation:

- Move GPU-Assisted validation into a standalone *ValidationObject*
- Implement per-API validation disables, perhaps through the configuration file
- Further cleanup on the CoreChecks *ValidationObject*, particularly the object state map accessor methods
- Better organize the *ValidationObject* and its child objects with regards to data hiding
- Implement per-object reader-writer locks to improve thread contention issues

References

Validation Layer Documents

Additional layer-specific information can be found in the KhronosGroup/Vulkan-ValidationLayers GitHub repository in the file docs directory:

- Vulkan-ValidationLayers/docs/khronos_validation_layer.md
- Vulkan-ValidationLayers/docs/core_validation_layer.md
- Vulkan-ValidationLayers/docs/object_tracker_layer.md
- Vulkan-ValidationLayers/docs/parameter_validation_layer.md
- Vulkan-ValidationLayers/docs/threading_layer.md
- Vulkan-ValidationLayers/docs/unique_objects_layer.md

A description of the options available in the layer configuration file can be found in the KhronosGroup/Vulkan-ValidationLayers GitHub repository in the layers directory:

- Vulkan-ValidationLayers/layers/vk_layer_settings.txt

The current layer description and status can be found in the KhronosGroup/Vulkan-ValidationLayers GitHub repository in the layers directory:

- Vulkan-ValidationLayers/layers/README.md

Validation Layer Source Codes

- The layer sources can be found in the KhronosGroup/Vulkan-ValidationLayers Github

Acknowledgements

Please acknowledge the extensive contributions of

- Dave Houlton, LunarG, Inc.
- Mike Schuchardt, LunarG, Inc.
- Jeremy Hayes, LunarG, Inc.
- Tobin Ehlis, Google, Inc.

Document Change Log

4/01/19 - Initial Revision