Vulkan Debug Utilities

A Vulkan Extension Tutorial
Mark Young, LunarG

Introduction

The Vulkan API is now two years old, and as with all things it is showing areas that require improvement. Debugging is one of the areas where we can make small changes that produce a large benefit for the Vulkan community. After soliciting input from IHVs and several game companies, and reviewing feedback from GitHub users, we decided to improve the debugging functionality exposed by both VK_EXT_debug_report and VK_EXT_debug_marker. As we investigated, we decided that replacing the extension was the proper decision going forward instead of trying to shoehorn new functionality into the existing extensions. The changes resulted in the creation of a new extension: VK_EXT_debug_utils.

Why the New Extension?

The Vulkan Working Group received feedback from developers at several software companies asking for more information from each debug message to help them isolate the trigger in their own code. Validation messages created a special concern since an application can create multiple Vulkan objects, and only one of those objects may be handled incorrectly.

To help software developers isolate issues more efficiently, LunarG decided to combine the functionality of VK_EXT_debug_report and VK_EXT_debug_marker to produce more useful debug messages. However, while attempting to coordinate the work between these two separate extensions, we recognized a fundamental problem. VK_EXT_debug_report is an instance extension, while VK_EXT_debug_marker is a device extension, and there is no easy and clean way to indicate that functionality in an instance extension is dependent upon a device extension being present and enabled. To simplify things, we decided we could just define a new instance extension that supplied all the necessary items in one place.

We also expanded the information that is returned to a user's debug callback. This change could have been made with the old extensions, but it would require adding items to the pNext chain of most structures. While doable, it added more complexity than we thought was worthwhile since every debug callback would have to care about the pNext chain. Of course, we still may add functionality to the new pNext chain in the future.

Finally, the VK_EXT_debug_report extension uses a special internal enumeration to track object types, VkDebugReportObjectTypeEXT. This enumeration was supported for a time, and even

used by the VK_EXT_debug_marker extension. However, the latest versions of the Vulkan spec replace this structure with a new core object type enumeration, VkObjectType. Due to this spec change, Khronos decided to stop expanding VkDebugReportObjectTypeEXT and instead support adding new enumeration values only to VkObjectType. Consequently, the VkDebugReportObjectTypeEXT enumeration will grow stale over time.

With all these factors under consideration, LunarG decided to create this new Vulkan debug utility from scratch.

Benefits of This New Extension

The VK_EXT_debug_utils introduces the concept of a debug messenger, VkDebugUtilsMessengerEXT. During creation, the application details what debug message types and severities are needed. Additionally, the application provides a function pointer to a callback message handler that should be triggered when a message with the appropriate severity and type is encountered. The VkDebugReportCallbackEXT object type works in a similar manner to the VK_EXT_debug_report object types. The improved usefulness resides in what data is now provided to the new callback. To read what data is provided to this callback, go to the "Creating a Debug Messenger Callback" section.

Like the VK_EXT_debug_marker extension, the new extension allows you to identify specific locations in a VkCommandBuffer. Previously, in the VK_EXT_debug_marker extension, the identified locations were called "markers." Now in VK_EXT_debug_utils, they are called "labels." In addition, VK_EXT_debug_utils adds the new ability to insert these "labels" into a VkQueue to indicate the progress of the runtime/driver/hardware in processing a Vulkan queue. For more information on "labels," refer to the "Adding Labels" section below.

Another feature of VK_EXT_debug_marker that is supported by the VK_EXT_debug_utils extension is the ability to associate application defined data with a Vulkan handle. The most common use case of this feature is naming each Vulkan handle with an easily identifiable string name. The handle value of a Vulkan object can change internal to any Vulkan component including the loader, layers, or ICD -- so without naming your handles, the information returned could be confusing. Here's an example:

> If you're calling vkCmdBindPipeline incorrectly, and you've enabled validation layers, you may find that the error message mentions a handle for the erroneous VkCommandBuffer. However, when you look for the problematic handle in your list of known VkCommandBuffer handles, you find that it doesn't match any known value, which could be caused by the loader or layer having their own handle for the same object.

However, if an application sets a name for each of its Vulkan handles, the names are associated with those handles in any component supporting this extension. This result is true even if the handle value for the object changes. In the above case, if you had named the VkCommandBuffer something like “Primary Command Buffer in Thread B,” you would get the unusual handle value back, but with the name “Primary Command Buffer in Thread B.” For more information on naming, refer to “Naming Objects.”

Additionally, VK_EXT_debug_utils continues to provide the ability to define object-specific binary content using a tag. The content of these tags tends to be very complex and is most often used for debugging layers that need the additional content, like RenderDoc. Tags are not used by any validation layer messages, and as such do not get returned to the user in the debug messenger callback. Refer to the “Tagging Objects” section below for more information.

After reading the information above, you may feel like the new extension doesn’t bring anything new to the table. However, if you look at the data returned in a callback registered with VK_EXT_debug_report, you’ll notice that when a debug message is returned to your callback, you only receive information about one object as well as the message. When using the new extension, VK_EXT_debug_utils combines most of the information you can set and passes this information to your callback function. Here is the additional information included in the callback message beyond what is returned by the old VK_EXT_debug_report extension:

- A list of relevant objects for each debug message
- A name associated with each object (if the name has been set)
- A list of command buffer labels encountered up to that point
    - Only if a VkCommandBuffer has been found in the object list and has labels associated with it
- A list of queue labels encountered up to that point
    - Only if a VkQueue has been found in the object list and has labels associated with it

With this additional information, you should find it easier to narrow down the location of a debug message in even the most complicated applications. This is especially useful if you have enabled VK_LAYER_LUNARG_standard_validation and you receive an error from validation messages about a particular sequence of commands.

How Do I Use It?

Before you can use it, you must make sure the VK_EXT_debug_utils extension is available using vkEnumerateInstanceExtensionProperties. But, once you have determined that the extension is available, you can unlock its debugging capabilities.

We cover the highlights of using VK_EXT_debug_utils below, but you can find detailed information about the use of this new extension in the "Debugging" section of the Vulkan specification.

Use this extension in a similar way to how you previously used the VK_EXT_debug_report and VK_EXT_debug_marker extensions. However, instead of enabling two extensions separately (one an instance extension, and the other a device extension), you enable a single instance extension.

First, we will discuss receiving debug messages using the new extension. If you don't need to receive debug messages, then you may skip to the "Naming Objects" section below.

Creating a Debug Messenger Callback

Since you intend to receive debug messages, you must first create a callback function that will receive debug messages and it must be formatted after the fashion of the PFN_vkDebugUtilsMessengerCallbackEXT function pointer.

```
typedef VkBool32 (VKAPI_PTR *PFN_vkDebugUtilsMessengerCallbackEXT)(
    VkDebugUtilsMessageSeverityFlagBitsEXT           messageSeverity,
    VkDebugUtilsMessageTypeFlagsEXT                  messageType,
    const VkDebugUtilsMessengerCallbackDataEXT*      pCallbackData,
    void*                                            pUserData);
```

You will notice that we've split the message severity (*messageSeverity*) from the message type (*messageType*) in the new callback. The severity indicates the importance of the message. The possible values are currently defined as:

```
typedef enum VkDebugUtilsMessageSeverityFlagBitsEXT {
    VK_DEBUG_UTILS_MESSAGE_SEVERITY_VERBOSE_BIT_EXT = 0x00000001,
    VK_DEBUG_UTILS_MESSAGE_SEVERITY_INFO_BIT_EXT    = 0x00000010,
    VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT = 0x00000100,
    VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT   = 0x00001000,
} VkDebugUtilsMessageSeverityFlagBitsEXT;
```

As the importance of the message increases, so does the enumeration value. In addition, we've left space for future types that could fit in between any of the existing values. For this reason, you can always compare the values in your callback:

```
if (messageSeverity >= VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT) {
        // This means the message was either a warning or error of some kind.
}
```

The message types describe what kind of message you are receiving. Currently, the following message types are available (however more could be added in the future):

```
typedef enum VkDebugUtilsMessageTypeFlagBitsEXT {
    VK_DEBUG_UTILS_MESSAGE_TYPE_GENERAL_BIT_EXT     = 0x00000001,
    VK_DEBUG_UTILS_MESSAGE_TYPE_VALIDATION_BIT_EXT  = 0x00000002,
    VK_DEBUG_UTILS_MESSAGE_TYPE_PERFORMANCE_BIT_EXT = 0x00000004,
} VkDebugUtilsMessageTypeFlagBitsEXT;
```

General messages typically come from Vulkan components themselves. The validation bit indicates that the message is related to the process of validating your application's behavior against the specification itself. These are the most common messages from the validation layers since most validation errors or warnings indicate a possible violation of the Vulkan specification. Finally, there are performance messages that suggest performance improvements you can make to your application.

pCallbackData points to a VkDebugUtilsMessengerCallbackDataEXT structure containing information about what triggered the message and enough help to figure out the location that caused the trigger:

```
typedef struct VkDebugUtilsMessengerCallbackDataEXT {
    VkStructureType                                 sType;
    const void*                                     pNext;
    VkDebugUtilsMessengerCallbackDataFlagsEXT       flags;
    const char*                                     pMessageIdName;
    int32_t                                         messageIdNumber;
    const char*                                     pMessage;
    uint8_t                                         queueLabelCount;
    VkDebugUtilsLabelEXT*                           pQueueLabels;
    uint8_t                                         cmdBufLabelCount;
    VkDebugUtilsLabelEXT*                           pCmdBufLabels;
    uint8_t                                         objectCount;
    VkDebugUtilsObjectNameInfoEXT*                  pObjects;
} VkDebugUtilsMessengerCallbackDataEXT;
```

The first three parameters in the structure are common to most Vulkan structures, so we won't discuss them here.

The second set of three parameters provides all the details about the specific message.

1. *pMessageIdName* is a string indicating what triggered the message. For validation layers, this string may contain the valid usage ID (VUID) string identifier that will help identify the specific portion of the specification that the layer believes was violated.
2. *messageIdNumber* indicates the unique number of this message (if it is non-zero). If this message was triggered by a validation layer, it will contain a unique numeric valid usage ID for the validation warning or error that occurred. Use this number in a table lookup to determine the location in the Vulkan specification that the validation layers believe you may have violated.
3. *pMessage* is a C-style string (null-terminated) that indicates the specifics of the message.

The callback returns both a message ID number and name. The validation layers currently return the *messageIdNumber* for a given validation message. If the *messageIdNumber* is present, you can look up the actual Valid Usage ID string by accessing the vk_validation_error_messages.h header file, finding the value in the UNIQUE_VALIDATION_ERROR_CODE enum, and then finding the value in the table mapping VUIDs to spec snippet. The spec snippet will contain the final VUID string that can be used to find the exact section of the spec. We realize this is a complex process for users, and going forward, the intent is that the validation layers will return the actual spec VUID string using the new p*MessageIdName* field.

Once you have the VUID string, you can open the Vulkan Specification and append the hash symbol (#) followed by the VUID string to jump directly to the section.

For example:

If you were able to determine that your VUID string was:

VUID-VkApplicationInfo-pApplicationName-parameter

You could access the spec section directly through:

https://www.khronos.org/registry/vulkan/specs/1.1/html/vkspec.html#VUID-VkApplicationInfo-pApplicationName-parameter

Now we get to items new to this extension. *queueLabelCount* and *pQueueLabels* contain any information about labels your application may have applied to any VkQueue. These fields will only be populated if a VkQueue object appears in the *pObjects* list. *pQueueLabels* will only contain labels set in the particular VkQueue object up to the point of the message being triggered. These labels contain the following information:

```
typedef struct VkDebugUtilsLabelEXT {
    VkStructureType    sType;
    const void*        pNext;
    const char*        pLabelName;
    float              color[4];
} VkDebugUtilsLabelEXT;
```

pLabelName is the name of the label your application defined and *color* is a floating point color you identify. While validation layers and debug messages don't use the color, other layers or even your own application could use this information.

If no VkQueue object appears in the *pObjects* list, then *queueLabelCount* should be 0 and *pQueueLabels* will be NULL. Items in the *pQueueLabels* array are sorted so that the most recent labels appear at the lower index values, i.e. the label at index 0 is the most recent label associated with the VkQueue.

Likewise, *cmdBufLabelCount* and *pCmdBufLabels* contain any labels from any VkCommandBuffer in the *pObjects* list. Labels can be inherited from a primary command buffer by a secondary command buffer. However, for debug messages, most layers and the loader only know information about the active command buffer, or its child objects. Therefore, *cmdBufLabelCount* will only be non-zero and *pCmdBufLabels* will only be non-NULL if any VkCommandBuffer appears in the *pObjects* list and that VkCommandBuffer has associated labels.

The final two elements of the VkDebugUtilsMessengerCallbackDataEXT are: *objectCount* and *pObjects*. *pObjects* contains the information about any objects that can be easily associated with a message. The information is stored in VkDebugUtilsObjectNameInfoEXT structures:

```
typedef struct VkDebugUtilsObjectNameInfoEXT {
    VkStructureType    sType;
    const void*        pNext;
    VkObjectType       objectType;
    uint64_t           objectHandle;
    const char*        pObjectName;
} VkDebugUtilsObjectNameInfoEXT;
```

Each object contained in *pObjects* will contain the object's type (*objectType*) and the object's handle (*objectHandle*). If you define a name for the object (as described in the "Naming Objects" section), the object's name (*pObjectName*) will be set to point to a string containing the name you provided -- allowing you to easily identify most objects. When combining use of *pObjects* with the labels, you should be able to narrow down what portion of your code triggered a message.

Finally, the callback function receives a pointer to user-supplied data (*pUserData*) that you provide to each messenger during its creation.

Creating (and Destroying) a Debug Messenger

Once you've setup your callback, you need to create a debug messenger that will be used to trigger the callback when a message occurs.

```
VkResult vkCreateDebugUtilsMessengerEXT(
    VkInstance                                instance,
    const VkDebugUtilsMessengerCreateInfoEXT* pCreateInfo,
    const VkAllocationCallbacks*              pAllocator,
    VkDebugUtilsMessengerEXT*                 pMessenger);
```

Again, this looks pretty standard for those of you who have done a lot of Vulkan work. The most important component is the 2nd parameter, *pCreateInfo*, which is a pointer to the following structure:

```
typedef struct VkDebugUtilsMessengerCreateInfoEXT {
    VkStructureType                     sType;
    const void*                         pNext;
    VkDebugUtilsMessengerCreateFlagsEXT  flags;
    VkDebugUtilsMessageSeverityFlagsEXT  messageSeverity;
    VkDebugUtilsMessageTypeFlagsEXT      messageType;
    PFN_vkDebugUtilsMessengerCallbackEXT pfnUserCallback;
    void*                               pUserData;
} VkDebugUtilsMessengerCreateInfoEXT;
```

The *messageSeverity* parameter is intended to indicate **all** message severities that you wish to trigger your callback. You'll notice that it uses the "Flags" and not "FlagBits" version of the severity because it may take more than one value. For example, you may set your messages severity as follows:

```
messageSeverity = VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT |
                  VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT;
```

Likewise, *messageType* is a combination of **all** message types you are interested in tracking. You then set *pfnUserCallback* with a pointer to your created callback function from above.

The messages that are returned to your callback must be of both a severity that you have enabled and a type that you have enabled during the vkCreateDebugUtilsMessengerEXT call.

Otherwise, if only the severity matches but not the type or vice-versa, it will not trigger your callback.

Finally, you can provide a pointer to additional data using *pUserData* or you can set it to NULL. Many times, applications pass a pointer to a structure or class that is used during your callback to perform some sort of logging.

As with all standard Vulkan objects, you destroy a VkDebugUtilsMessengerEXT with a call to vkDestroyDebugUtilsMessengerEXT.

```
void vkDestroyDebugUtilsMessengerEXT(
    VkInstance                   instance,
    VkDebugUtilsMessengerEXT     messenger,
    const VkAllocationCallbacks* pAllocator);
```

Naming Objects

Naming allows an application to identify objects using a specific name and is useful because object handle values can (and do) change when entering the loader, a layer, or even a runtime. If a message occurs in one of those layers for a particular object, the handle will be unknown to the user and could cause confusion. Thus naming objects was born.

Let's take a quick look at why naming is so useful. Say you have an application making an incorrect call that is caught by the validation layers. Let's also say, that the Vulkan loader is wrapping the object itself so that it can keep track of other data associated with it. At the application you may have a handle of 0xFEED for your object. The loader, however, unwraps that information, and now passes 0xF00D down to the first layer. That layer also unwraps it and it now becomes 0xBEEF. Finally, the validation layer discovers the bug and let's you know (using your Debug Utils Messenger callback) that 0xBEEF was wrong. But, you wonder, what is object 0xBEEF?

Now, if you had "named" the object something useful, it would be easy to identify. Let's go back to the above example. Instead, before you make your call, you identify the name "Hamburger" with your object you associated with 0xFEED. If the loader encounters a bug, you get back 0xF00D as a handle, but also a name "Hamburger." If one of the layers encounter the bug, you get 0xBEEF, but also with "Hamburger." Obviously, naming is much more helpful.

Naming objects with `VK_EXT_debug_utils` is similar to the way you name objects with `VK_EXT_debug_marker`. You simply make a call to:

```
VkResult vkSetDebugUtilsObjectNameEXT(
    VkDevice                                    device,
    const VkDebugUtilsObjectNameInfoEXT* pNameInfo);
```

The structure is very similar to the VkDebugMarkerObjectNameInfoEXT found in the VK_EXT_debug_marker extension. The main difference is simply the member names, and the fact that the type in the new VkDebugUtilsObjectNameInfoEXT structure uses the VkObjectType enum instead of the VkDebugReportObjectTypeEXT enum. Notice that this is the same exact structure returned to the callback as described in "Creating (and Destroying) a Debug Messenger," but we'll restate it here just so you can see the format.

```
typedef struct VkDebugUtilsObjectNameInfoEXT {
    VkStructureType    sType;
    const void*        pNext;
    VkObjectType       objectType;
    uint64_t           objectHandle;
    const char*        pObjectName;
} VkDebugUtilsObjectNameInfoEXT;
```

Tagging Objects

Tagging is similar to naming, but has a very different purpose. For naming, you associate an application provided string to a particular object. Tagging, instead, associates an integer ID and binary data with an object. For the best use of tagging, both the application and the layer and/or runtime must agree on the meaning of the IDs and the use of the data.

Examples of tagging might include:

- Tagging a shader object with the human-readable vertex and fragment shader contents
- Tagging a buffer with either the contents or metadata about the contents

Typically, tagging is used for debug or performance tools such as RenderDoc.

In general, most debug layers and tools get the necessary information they need by simply referring to objects by name. Since the Debug Utils Messenger callbacks return to the application, no binary data is needed to pass through the layers except a name to keep track of unique objects. The application is already the keeper of all the information it needs, so no tagging information is returned through the Debug Utils Messenger callback.

However, if you need to use tagging, VK_EXT_debug_utils implements it in a similar way as VK_EXT_debug_marker. In this case, you now call:

```
VkResult vkSetDebugUtilsObjectTagEXT(
    VkDevice                                    device,
    const VkDebugUtilsObjectTagInfoEXT* pTagInfo);
```

The structure is also similar to the VkDebugMarkerObjectTagInfoEXT found in the VK_EXT_debug_marker extension. Two differences are: 1) the member names, and 2) the type in the new VkDebugUtilsObjectNameInfoEXT structure uses the VkObjectType enum instead of the VkDebugReportObjectTypeEXT enum:

```
typedef struct VkDebugUtilsObjectTagInfoEXT {
    VkStructureType    sType;
    const void*        pNext;
    VkObjectType       objectType;
    uint64_t           objectHandle;
    uint64_t           tagName;
    size_t             tagSize;
    const void*        pTag;
} VkDebugUtilsObjectTagInfoEXT;
```

Some of you may be unfamiliar with this structure, so I'll briefly cover it. *objectType* and *objectHandle* are exactly the same as naming. *tagName* is a numeric name or identifier for this tag and should be used to indicate the type of data being tagged. Again, if you implement a layer to intercept this information, this value would indicate the specific information you're attempting to set for the indicated object. *pTag* is a pointer to data of *tagSize* bytes that is being associated with this object.

Adding Labels

Sometimes, even knowing what object encountered a problem isn't enough. Often, you may touch an object many times and in similar ways throughout a frame. In these cases, it would be great if you could narrow down where in the frame (or even multiple frames) an issue occurred -- like how there are mile-markers set along the side of a highway to let you know where you are when driving. Labels are a great way to insert custom identifiers along the way to help you identify where you are at any given time.

Using `VK_EXT_debug_utils` you can insert labels into either a VkQueue, or a VkCommandBuffer, similarly to "markers" first exposed by the `VK_EXT_debug_marker` extension -- except you could only add markers to a VkCommandBuffer object.

There are two methods for adding labels to either a VkQueue or a VkCommandBuffer:

- Starting and stopping a label region
- Simply inserting a label

Let's look at a quick example to understand how we would use both.

Let's say you're drawing a humanoid figure for your application. You might draw it in the following way:

```
DrawFigure()
{
    UpdateMatrix();
    DrawUpperBody();
    DrawLowerbody();
}

DrawUpperBody()
{
    UpdateMatrix();
    DrawChest();
    DrawHeadNeck();
    DrawLeftArm();
    DrawRightArm();
}

DrawLowerBody()
{
    UpdateMatrix();
    DrawWaist();
    DrawLeftLeg();
    DrawRightLeg();
}
```

Running validation, you notice an error on a draw. But which draw and how do you narrow it down? With labels, you could do the following:

```
DrawFigure()
{
    BeginLabel("Draw Figure");
    UpdateMatrix();
    DrawUpperBody();
    DrawLowerbody();
    EndLabel();  // "Draw Figure"
}

DrawUpperBody()
{
    BeginLabel("Draw Upper Body");
    UpdateMatrix();
    DrawChest();
    InsertLabel("Draw head and neck");
    DrawHeadNeck();
    InsertLabel("Draw left arm");
    DrawLeftArm();
    InsertLabel("Draw right arm");
    DrawRightArm();
    EndLabel();  // "Draw Upper Body"
}

DrawLowerBody()
{
    BeginLabel("Draw Lower Body");
    UpdateMatrix();
    InsertLabel("Draw waist");
    DrawWaist();
    InsertLabel("Draw left leg");
    DrawLeftLeg();
    InsertLabel("Draw right leg");
    DrawRightLeg();
    EndLabel();  // "Draw Lower Body"
}
```

Both the begin/end and inserts are used in ways that allow you to easily identify a particular region. Now, if an error or warning occurred during the “DrawLeftLeg” routine, you would know which draw to target.

Labels can be added on a either a command buffer or a queue basis. To begin a label region in a queue, you use the following command:

```
void vkQueueBeginDebugUtilsLabelEXT(
    VkQueue                      queue,
    const VkDebugUtilsLabelEXT*  pLabelInfo);
```

This command accepts the following structure:

```
typedef struct VkDebugUtilsLabelEXT {
    VkStructureType    sType;
    const void*        pNext;
    const char*        pLabelName;
    float              color[4];
} VkDebugUtilsLabelEXT;
```

This is the same exact structure returned to the callback as described in “Creating (and Destroying) a Debug Messenger.” *Color* may be used by a layer or tool to generate colored text for the label name. If you don’t care about the color, set each value of this parameter “1.0” in case a tool does use the color.

Once you finish that labeled section, you call:

```
void vkQueueEndDebugUtilsLabelEXT(
    VkQueue  queue);
```

Notice that it doesn’t take the label again. In this way, the end works similar to a “pop” command by simply ending the last label that was created on that particular VkQueue with the vkQueueBeginDebugUtilsLabelEXT command.

If, instead of defining a whole region, you can insert a single label identifying a particular location, use:

```
void vkQueueInsertDebugUtilsLabelEXT(
    VkQueue                      queue,
    const VkDebugUtilsLabelEXT*  pLabelInfo);
```

If you define labels associated with a VkQueue and a message is triggered that has that particular VkQueue in the list of objects, then it will also populate the *queueLabelCount* and *pQueueLabels* data fields of the callback with the appropriate content. In that case, the data will be stored similar to a stack, with the first element being the most recent label and the last element being the oldest.

Likewise, you can insert labels inside a command buffer using commands similar to the above VkQueue commands:

```
void vkCmdBeginDebugUtilsLabelEXT(
    VkCommandBuffer             commandBuffer,
    const VkDebugUtilsLabelEXT* pLabelInfo);
void vkCmdEndDebugUtilsLabelEXT(
    VkCommandBuffer             commandBuffer);
void vkCmdInsertDebugUtilsLabelEXT(
    VkCommandBuffer             commandBuffer,
    const VkDebugUtilsLabelEXT* pLabelInfo);
```

Notice how similar they are. The only real difference being that they require a VkCommandBuffer instead of a VkQueue. Like the VkQueue labels, if a Debug Utils Messenger callback is triggered with a VkCommandBuffer object in the pObjects list and that command buffer contains labels, the labels will be added to the *cmdBufLabelCount* and *pCmdBufLabels* fields of the callback. Again, like the queue content, the data is returned similar to a stack, with the first label being the most recent and the last label being the oldest. Let's look at our example about the draws. If the bug was in the DrawLeftLeg() routine, then when the error returned, the values of *cmdBufLabelCount* and *pCmdBufLabels* would look like the following:

```
cmdBufLabelCount = 3;
pCmdBufLabels[0] = "Draw left leg";
pCmdBufLabels[1] = "Draw Lower Body";
pCmdBufLabels[2] = "Draw Figure";
```

One special attribute of command buffer labels is that they can be enabled/disabled across command buffer boundaries. For example, you may begin a label in a primary command buffer and end it in a secondary command buffer. Or you could begin a label in one secondary command buffer and end it in an entirely different secondary command buffer. The only caveat is that the command buffer dependency chain may not be known at the time validation (and other) messages are triggered. Therefore, for those kind of messages you can only count on the contents of the one command buffer. However, for tools like RenderDoc, the spanning of command buffers should function correctly.

Application Usage Examples

Setting Up a Debug Utils Messenger and Callback

The following code snippet shows how to setup a Debug Utils Messenger callback and create a messenger that will use it. Also, notice that because this is an extension, you need to query the Vulkan commands for VK_EXT_debug_utils using the vkGetInstanceProcAddr:

```
PFN_vkCreateDebugUtilsMessengerEXT CreateDebugUtilsMessengerEXT;
PFN_vkDestroyDebugUtilsMessengerEXT DestroyDebugUtilsMessengerEXT;
VkDebugUtilsMessengerEXT dbg_messenger;
VkInstance instance;

// Define a callback to capture the messages
VKAPI_ATTR VkBool32 VKAPI_CALL debug_messenger_callback(
    VkDebugUtilsMessageSeverityFlagBitsEXT           messageSeverity,
    VkDebugUtilsMessageTypeFlagsEXT                  messageType,
    const VkDebugUtilsMessengerCallbackDataEXT*      callbackData,
    void*                                            userData) {
    char prefix[64];
    char *message = (char *)malloc(strlen(callbackData->pMessage) + 500);
    assert(message);

    if (messageSeverity & VK_DEBUG_UTILS_MESSAGE_SEVERITY_VERBOSE_BIT_EXT) {
        strcpy(prefix, "VERBOSE : ");
    } else if (messageSeverity & VK_DEBUG_UTILS_MESSAGE_SEVERITY_INFO_BIT_EXT) {
        strcpy(prefix, "INFO : ");
    } else if (messageSeverity & VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT) {
        strcpy(prefix, "WARNING : ");
    } else if (messageSeverity & VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT) {
        strcpy(prefix, "ERROR : ");
    }

    if (messageType & VK_DEBUG_UTILS_MESSAGE_TYPE_GENERAL_BIT_EXT) {
        strcat(prefix, "GENERAL");
    } else {
        if (messageType & VK_DEBUG_UTILS_MESSAGE_TYPE_SPECIFICATION_BIT_EXT) {
            strcat(prefix, "SPEC");
            validation_error = 1;
        }
        if (messageType & VK_DEBUG_UTILS_MESSAGE_TYPE_PERFORMANCE_BIT_EXT) {
            if (messageType & VK_DEBUG_UTILS_MESSAGE_TYPE_SPECIFICATION_BIT_EXT) {
                strcat(prefix, "|");
            }
            strcat(prefix, "PERF");
        }
```

```
    }

    sprintf(message,
            "%s - Message ID Number %d, Message ID String :\n%s",
            prefix,
            callbackData->messageIdNumber,
            callbackData->pMessageIdName,
            callbackData->pMessage);
    if (callbackData->objectCount > 0) {
        char tmp_message[500];
        sprintf(tmp_message, "\n   Objects - %d\n", callbackData->objectCount);
        strcat(message, tmp_message);
        for (uint32_t object = 0; object < callbackData->objectCount; ++object) {
            sprintf(tmp_message,
                    "       Object[%d] - Type %s, Value %p, Name \"%s\"\n",
                    Object,
                    DebugAnnotObjectToString(
                        callbackData->pObjects[object].objectType),
                    (void*)(callbackData->pObjects[object].objectHandle),
                    callbackData->pObjects[object].pObjectName);
            strcat(message, tmp_message);
        }
    }
    if (callbackData->cmdBufLabelCount > 0) {
        char tmp_message[500];
        sprintf(tmp_message,
                "\n   Command Buffer Labels - %d\n",
                callbackData->cmdBufLabelCount);
        strcat(message, tmp_message);
        for (uint32_t label = 0; label < callbackData->cmdBufLabelCount; ++label) {
            sprintf(tmp_message,
                    "       Label[%d] - %s { %f, %f, %f, %f}\n",
                    Label,
                    callbackData->pCmdBufLabels[label].pLabelName,
                    callbackData->pCmdBufLabels[label].color[0],
                    callbackData->pCmdBufLabels[label].color[1],
                    callbackData->pCmdBufLabels[label].color[2],
                    callbackData->pCmdBufLabels[label].color[3]);
            strcat(message, tmp_message);
        }
    }

    printf("%s\n", message);
    fflush(stdout);
    free(message);

    // Don't bail out, but keep going.
    return false;
}
```

```
// Setup our pointers to the VK_EXT_debug_utils commands
CreateDebugUtilsMessengerEXT =
    (PFN_vkCreateDebugUtilsMessengerEXT)vkGetInstanceProcAddr(
        instance,
        "vkCreateDebugUtilsMessengerEXT");
DestroyDebugUtilsMessengerEXT =
    (PFN_vkDestroyDebugUtilsMessengerEXT)vkGetInstanceProcAddr(
        instance,
        "vkDestroyDebugUtilsMessengerEXT");

// Create a Debug Utils Messenger that will trigger our callback for any warning
// or error.
VkDebugUtilsMessengerCreateInfoEXT dbg_messenger_create_info;
dbg_messenger_create_info.sType =
VK_STRUCTURE_TYPE_DEBUG_UTILS_MESSENGER_CREATE_INFO_EXT;
dbg_messenger_create_info.pNext = NULL;
dbg_messenger_create_info.flags = 0;
dbg_messenger_create_info.messageSeverity =
    VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT |
    VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT;
dbg_messenger_create_info.messageType =
    VK_DEBUG_UTILS_MESSAGE_TYPE_GENERAL_BIT_EXT |
    VK_DEBUG_UTILS_MESSAGE_TYPE_SPECIFICATION_BIT_EXT |
    VK_DEBUG_UTILS_MESSAGE_TYPE_PERFORMANCE_BIT_EXT;
dbg_messenger_create_info.pfnUserCallback = debug_messenger_callback;
dbg_messenger_create_info.pUserData = NULL;
VkResult result = CreateDebugUtilsMessengerEXT(instance,
                                               &dbg_messenger_create_info,
                                               NULL,
                                               &dbg_messenger);

// Do all your stuff

// Destroy the Debug Utils Messenger
DestroyDebugUtilsMessengerEXT(instance, dbg_messenger, NULL);
```

Using Object Names and Command Buffer Labels

The following code snippet shows how to define a label and a named object:

```
PFN_vkCmdBeginDebugUtilsLabelEXT CmdBeginDebugUtilsLabelEXT;
PFN_vkCmdEndDebugUtilsLabelEXT CmdEndDebugUtilsLabelEXT;
PFN_vkCmdInsertDebugUtilsLabelEXT CmdInsertDebugUtilsLabelEXT;
PFN_vkSetDebugUtilsObjectNameEXT SetDebugUtilsObjectNameEXT;
VkInstance instance;
VkDevice device;
VkCommandBuffer cmd_buf;
VkDebugUtilsLabelEXT label;
```

```
// Setup the function pointers
CmdBeginDebugUtilsLabelEXT =
    (PFN_vkCmdBeginDebugUtilsLabelEXT)vkGetInstanceProcAddr(
        instance,
        "vkCmdBeginDebugUtilsLabelEXT");
CmdEndDebugUtilsLabelEXT =
    (PFN_vkCmdEndDebugUtilsLabelEXT)vkGetInstanceProcAddr(
        instance,
        "vkCmdEndDebugUtilsLabelEXT");
CmdInsertDebugUtilsLabelEXT =
    (PFN_vkCmdInsertDebugUtilsLabelEXT)vkGetInstanceProcAddr(
        instance,
        "vkCmdInsertDebugUtilsLabelEXT");
SetDebugUtilsObjectNameEXT =
    (PFN_vkSetDebugUtilsObjectNameEXT)vkGetInstanceProcAddr(
        instance,
        "vkSetDebugUtilsObjectNameEXT");

// ...
vkBeginCommandBuffer(cmd_buf, &cmd_buf_info);

// Set a name for the command buffer
VkDebugUtilsObjectNameInfoEXT cmd_buf_name = {
    .sType = VK_STRUCTURE_TYPE_DEBUG_UTILS_OBJECT_NAME_INFO_EXT,
    .pNext = NULL,
    .objectType = VK_OBJECT_TYPE_COMMAND_BUFFER,
    .objectHandle = (uint64_t)cmd_buf,
    .pObjectName = "CubeDrawCommandBuf",
};
SetDebugUtilsObjectNameEXT(device, &cmd_buf_name);

// Begin a label section indicating we're in a draw.
label.sType = VK_STRUCTURE_TYPE_DEBUG_UTILS_LABEL_EXT;
label.pNext = NULL;
label.pLabelName = "Inside Draw";
label.color = {0.4f, 0.3f, 0.2f, 0.1f};
CmdBeginDebugUtilsLabelEXT(cmd_buf, &label);

// Do other stuff

// Insert a single label
label.pLabelName = "Temp Label";
label.color = {1.0f, 1.0f, 1.0f, 1.0f};
CmdInsertDebugUtilsLabelEXT(cmd_buf, &label);

// End the label (this rolls back to any label before "Inside Draw")
// since that is the last one that was created with CmdBeginDebugUtilsLabelEXT
CmdEndDebugUtilsLabelEXT(cmd_buf);
```

Updates to the Cube Demo

The LunarG Cube demo has been updated to use the new VK_EXT_debug_utils extension callbacks when validation is enabled. Additionally, labels have been added to the main rendering command buffer and several of the objects. This should now provide a good example for you to use as a basis for implementing the changes in your own application.

vkCmdBindPipeline Error Output:

In order to show some example output, I commented out the vkCmdBindPipeline in the newly modified Cube demo and ran it with validation layers enabled (--validate). This was the result for both the old and new extension callbacks:

Old Method (VK_EXT_debug_report):

```
ERROR: [DS] Code 7 : Object: 0x1482130 (Name = CubeDrawCommandBuf) | At Draw/Dispatch
time no valid VkPipeline is bound! This is illegal. Please bind one with
vkCmdBindPipeline().
```

Some of you may notice that there's actually a little more information here than normal. The two most important non-NULL Vulkan objects (Object[0] and Object[1]) are now added to the front of the error message. In addition, if either of them has a name, it is also added. The error message is a little more lengthy, but it provides some of the information also provided in the new extension to help those of you who continue to use the older extension.

New Method (VK_EXT_debug_utils):

```
ERROR : VALIDATION - Message Id Number: 7 | Message Id Name: DS
        At Draw/Dispatch time no valid VkPipeline is bound! This is illegal. Please bind
one with vkCmdBindPipeline().

        Objects - 1
                Object[0] - VkCommandBuffer, Handle 0x1482130, Name "CubeDrawCommandBuf"

        Command Buffer Labels - 3
                Label[0] - ActualDraw { -0.400000, -0.300000, -0.200000, -0.100000}
                Label[1] - InsideRenderPass { 8.400000, 7.300000, 6.200000, 7.100000}
                Label[2] - DrawBegin { 0.400000, 0.300000, 0.200000, 0.100000}
```

The formatting is from the callback function and only the command buffer is named, but you can see the labels at the end (ordered in priority, similar to a stack). From this, you can see that the VkCommandBuffer named "CubeDrawCommandBuf" failed to bind a VkPipeline by the time we hit the "ActualDraw" label. With this, we believe you can quickly find out where your error in code resides.

Layout Warning/Error Output:

Another change shows both a warning and an error. In this case, cube.c was modified so that in the demo_prepare_texture_image() function, initialization of the VkImageCreateInfo structure was removed in the following way:

```
    const VkImageCreateInfo image_create_info = {
        .sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO,
        .pNext = NULL,
        .imageType = VK_IMAGE_TYPE_2D,
        .format = tex_format,
        .extent = {tex_width, tex_height, 1},
        .mipLevels = 1,
        .arrayLayers = 1,
        .samples = VK_SAMPLE_COUNT_1_BIT,
        .tiling = tiling,
        .usage = usage,
        .flags = 0,
// Disable        .initialLayout = VK_IMAGE_LAYOUT_PREINITIALIZED,
    };
```

This produced the following output:

Old Method (VK_EXT_debug_report):

```
WARNING: [DS] Code 6 : Object: 0x14 | Mapping an image with layout
VK_IMAGE_LAYOUT_UNDEFINED can result in undefined behavior if this memory is used by the
device. Only GENERAL or PREINITIALIZED should be used.
ERROR: [DS] Code 6 : Object: 0x28fe8e0 | Cannot submit cmd buffer using image (0x13)
[sub-resource: aspectMask 0x1 array layer 0, mip level 0], with layout
VK_IMAGE_LAYOUT_UNDEFINED when first use is VK_IMAGE_LAYOUT_PREINITIALIZED.
```

Again, notice how informative, yet cluttered, the information appears. Similarly, notice that in this case, we didn't name the command buffer, so no name appears. Now let's look at the new method:

New Method (VK_EXT_debug_utils):

```
WARNING : VALIDATION - Message Id Number: 6 | Message Id Name: DS
       Mapping an image with layout VK_IMAGE_LAYOUT_UNDEFINED can result in undefined
behavior if this memory is used by the device. Only GENERAL or PREINITIALIZED should be
used.

       Objects - 1
              Object[0] - VkDeviceMemory, Handle 0x14

ERROR : VALIDATION - Message Id Number: 6 | Message Id Name: DS
       Cannot submit cmd buffer using image (0x13) [sub-resource: aspectMask 0x1 array
layer 0, mip level 0], with layout VK_IMAGE_LAYOUT_UNDEFINED when first use is
VK_IMAGE_LAYOUT_PREINITIALIZED.

       Objects - 1
              Object[0] - VkCommandBuffer, Handle 0x28fe8e0
```

In this case, you can also see that there are no command buffer labels for this command buffer. So, it's obviously a different command buffer than we used the first time (in addition to noticing the missing name).

Future Validation Layer Improvements

As of the publication of this tutorial, the validation layers will return roughly the same information for both the VK_EXT_debug_report extension and the VK_EXT_debug_utils extension callbacks. The extensions differ in that the VK_EXT_debug_report extension will also automatically add in VkCommandBuffer and VkQueue labels to the callback results and separate out some of the information so that the returned messages don't seem as cluttered. Going forward, we will be adding additional object information to those error messages that we determine could benefit from such information. If there is a validation error message you feel deserves additional object information, please feel free to submit a GItHub Issue stating clearly which message and what object information you would like added. Or even better, you can supply your own GitHub Pull Request with the completed change.

Conclusion

As you can see, the VK_EXT_debug_utils extension brings some much needed debug information to the validation layers. We've added support for the VK_EXT_debug_utils extension starting with our Vulkan 1.1-capable loader. You don't need to use Vulkan 1.1 to expose the extension functionality as the release of this extension and Vulkan 1.1 just happen to be timed together. If you desire to use this extension in any fashion, please download

LunarG's Vulkan 1.1 SDK for your system. We hope you find the VK_EXT_debug_utils extension as easy to use as it is powerful.

Acknowledgements

I'd like to thank Erika Johnson, Mark Lobodzinski, and Mike Schuchardt from LunarG for helping to put the finishing touches on this article. Also, thanks to Piers Daniel at NVIDIA for giving it a final once over.