AN13778

Porting VGLite Driver for Bare Metal or Single Task

Rev. 0 — 18 November 2022 Application note

Document information

Information	Content
Keywords	VGLite, bare metal, single task
Abstract	The VGLite middleware includes VGLite driver, font and text draw API, and VGLite Elementary Library.

1 Introduction

The VGLite middleware includes VGLite driver, font and text draw API, and VGLite Elementary Library. The VGLite driver provides a set of native APIs that supports 2D vector-based and 2D raster-based operations. It can be used as the interface to 2D GPU hardware in the NXP i.MX RT500, i.MX RT1160, and i.MX RT1170 series chips. Font and text draw API and VGLite Elementary Library are based on the native API. All API and library in the VGLite middleware are platform independent and the implementation in the NXP MCUXpresso SDK is only for FreeRTOS.

This document:

- outlines the middleware components and driver architecture, including folder hierarchy and brief description of each folder.
- analyzes how driver supports multiple tasks, especially command buffers management to support multi-task.
- gives the driver porting for bare metal in details.
- provides the porting for RTOS to support single task.

The *VGLite Driver Porting Guide* (document IMXRTVGLITEPG) provides the most information about porting the VGLite driver to a specific OS platform. This document extends Its content with the analysis of command buffers management and command submission to GPU hardware.

The audience is encouraged to read *VGLite Driver Porting Guide* (document IMXRTVGLITEPG) before starting this document.

2 VGLite middleware architecture

The VGLite middleware is in the *middleware\vglite* folder under SDK installation root. It consists of the following components:

- VGLite native API
- Elementary library
- Font and text support

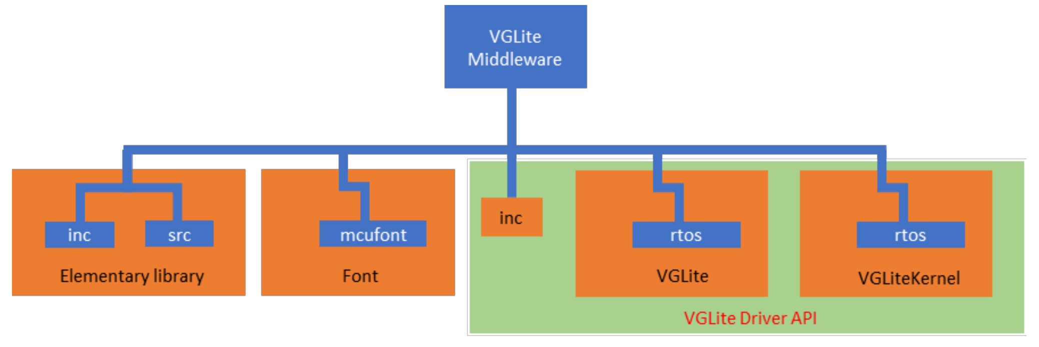

Figure 1. VGLite middleware architecture

2.1 VGLite native API

The VGLite API driver is the core of VGLite middleware. It provides a feature set smaller than that of OpenVG with maximum 2D vector/raster rendering performance but having minimum memory footprint. It is used for the application in the embedded system where memory size is limited. For detailed API information, refer to *i.MX RT VGLite API Reference Manual* (document IMXRTVGLITEAPIRM).

The VGLite driver consists of the following components:

- User API functions
  - OS independent layer
  - OS-specific layer
- Kernel driver
  - OS independent HAL layer
  - OS-specific layer

2.1.1 User API

The User API includes a set of API functions, type and structure definitions for parameters, and related enumerations. All are defined in a header file `vg_lite.h`. Application must include this file in the project to use 2D GPU for vector and/or raster operations.

To easily port the driver to a new specific OS, the code is layered. The OS independent layer implements the main functions of the driver. It hardly needs any modification while porting it to a new OS platform.

The OS-specific layer of user API provides the OS abstract layer executed in the use space. It supports with OS services that include mutexes and semaphores synchronization objects, VGLite commands queue, multiple tasks maintenance and management, and OS-specific memory management for application. In general, these codes are OS specific and executed in the user space if OS separates user space and kernel space.

2.1.2 Kernel driver

The VGLite kernel driver is expected to execute in the kernel space. It receives the request from VGLite User API to manipulate the GPU hardware to do corresponding operations and/or provide other OS kernel-specific services. To port the kernel driver to a new OS, it is also well layered.

The OS independent Hardware Abstract Layer (HAL) abstracts the GPU hardware-related common operations for kernel driver. These operations include initialization and de-initialization of GPU hardware to allocate/free OS kernel resources for GPU, memory allocation and free for the physical memory preserved for VGLite application, memory map and unmap for common physical memory, GPU register access, GPU reset, and synchronization between CPU and GPU.

The OS-specific kernel driver is like that of User API. It provides the OS services required by VGLite kernel driver, such as synchronization objects, OS memory management. If an OS running is separated into user space and kernel space, execute these codes in the kernel space. In another RTOS, for instance, FreeRTOS, there is no separation between user space and kernel space. The OS-specific layer of kernel driver and that of user API can share the implementation.

2.2 Elementary library

The VGLite driver API provides a set of primitive interfaces to render both vector graphics and images. Using these APIs to draw the graphic resources of application, developer must elaborately define the path data that includes a sequence of segment commands and their coordinates to represent the expected shape of the graphic resources in the application. It is not easy to manually control the layout of the geometry in the application that will be rendered using the primitive data. It also takes quite a bit effort to render these graphics as the expected appearance in a GUI of application.

To eliminate the effort of the developer to render the complex graphics, the Elementary library is provided. Elementary API is more effective and able to produce higher quality output than VGLite driver API. It wraps the VGLite API but provides a simple and intuitive method to load up and manipulate the graphic resources based on the three types of specific Elementary objects. The VGLite Toolkit provides a set of tools to convert specific SVG file or raster image into these objects. For details about VGLite Toolkit, see *VGLite Toolkit user guide*.

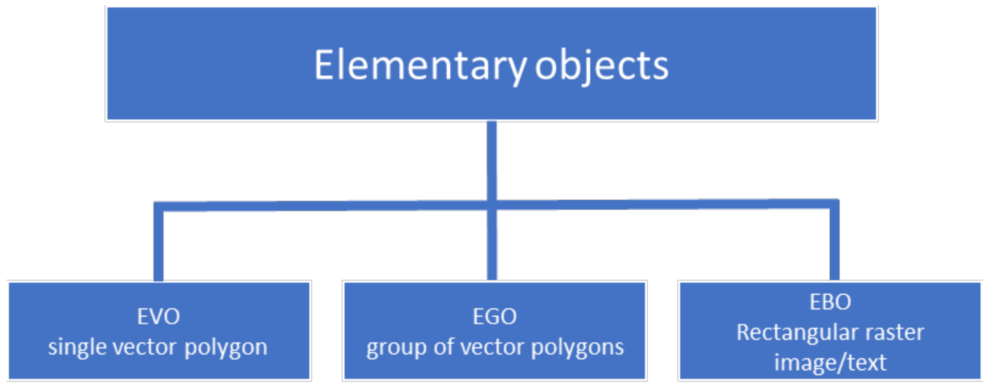

Figure 2. Elementary objects

The OS-specific code of Elementary library is in the `elementary\src\elm_os.c`. The only one routine is to implement the local storage of a task.

2.3 Font and Text support

Although VGLite hardware does not support to draw text using various fonts, VGLite middleware implements a prototype of font and text support as a software extension of VGLite. The vector font data is created by `vft_create` from an SVG file which contains an embedded font. Only SVG file that defines the font glyphs using the `font` and `glyph` objects can be supported in current implementation.

3 Support multiple tasks

VGLite driver API can be used simultaneously by different tasks. Each task uses hardware as if only itself is using the hardware without concern on any other task who is using GPU hardware at the same time. This benefit is from the VGLite driver implementation to support multiple instances.

VGLite driver API uses some mechanisms including task local storage, synchronization objects (Event, Mutex, or notification), and command buffer queue to guarantee multiple tasks simultaneous working.

The command queue task is created in the VGLite driver to implement one hardware to serve multiple tasks. Tasks submit their command into the command queue. Command queue task is responsible for taking a command away from the queue and submitting it into the GPU hardware to execute.

3.1 Task local storage

To support multiple tasks drawing, each task must create/destroy its drawing context using the `vg_lite_init` / `vg_lite_close` API. The drawing context contains all information that both driver API and hardware will use to operate a command. To maintain the drawing context for a task, each task allocates a buffer to save its context when the drawing context is created and frees the buffer when the drawing context is destroyed.

Different OS has its mechanism to implement a task local storage for above purpose. With task local storage, a task can correctly submit a command to hardware. Then hardware can process the command to finish the vector graphic drawing or render a raster image texture.

3.2 Synchronization

In multiple tasks environment, two or more tasks are likely submitting their command buffers at the same time. And a task may submit a new command buffer but the previous submitted command buffer likely is not finished processing or even still in the queue and not processed by hardware at all. All these cases need synchronization objects to have VGLite driver correctly communicate between tasks and/or with command queue.

Message queue supported by almost OS is used to synchronize the command buffers submit from different tasks. To simplify command management of a task and minimize the buffer use, a task uses two command buffers in ping-pang mode. And a task is only allowed to submit one command buffer into the command queue. The task must wait for the notification sent by command queue task to know that the hardware processes the previous submitted command buffer. After receiving the notification, the waiting task can submit a new command buffer into the command queue. In driver implementation, an event is created and bound with command buffers of a task and is used for this synchronization.

After a task submits a command buffer into the command queue, it does not need to wait this command buffer proceeded other than does something else such as preparation for the next command of drawing. The submitted command buffer is finally taken away from the command queue and submitted into GPU hardware to execute by command queue task. Command queue task then waits for this command done by hardware by waiting for a notification. When GPU hardware finished the command, an interrupt of GPU is fired that triggers the GPU ISR to send the notification. An event is used for this notification in the VGLite driver.

3.3 Command buffer management

To support multiple tasks, the VGLite driver is designed to use a queue to maintain the command buffers submitted from different tasks. Each task submits the command buffer after it is ready for GPU hardware to process. The different tasks may have different priorities but commands buffers from different tasks are maintained in the queue with FIFO mode. The command buffer submitted first into the queue is first taken away from the queue and sent hardware to process. With this method, a task feels like it is occupying GPU alone. There are three stages for a command buffer in the VGLite driver:

1. Commands of hardware operations and related data are assembled into a buffer when a task calls VGLite APIs to draw vector graphics or render an image. At last driver submits this command buffer into a command buffer queue by calling the queue write method serviced by OS.

2. Submitted command buffer is appended at the end of the queue. The command queue task takes the command buffer away from the head of the queue and sends it to GPU hardware to process.
3. Commands and related data in the command buffer are processed by GPU hardware. An interrupt is generated after all commands in the buffer are processed.

VGLite driver uses two command buffers in ping-pang mode for each task, creates a synchronization object, and binds it with command buffers. An application calls VGLite APIs to write the commands and necessary data into a command buffer, then submit it for hardware process. Driver does not wait for the completion of the currently submitted command buffer processed by hardware but waits for the completion of the previously submitted command buffer processed by hardware. As a result, only one command buffer is submitted to process at any time in a task. Multiple tasks use the same way to submit their command buffer as if each task uses the hardware alone without aware of other tasks.

The command buffer submitted by a task is not immediately submitted into hardware for process. Instead, it is first written to the tail of the queue that maintains all command buffers submitted from different tasks with FIFO mode. VGLite driver uses a command-queue task to take a command buffer from the head of the queue and send it into hardware to process. The command-queue task works in synchronization mode to wait for the notification of command buffer processing done after it sends a command buffer into hardware. When command-queue task receives the notification from GPU ISR to indicate that the just submitted command buffer has been processed, it sends another notification by triggering the synchronization object that is bound to the processed command buffer to wake up the task who is waiting for the completion of this command buffer processing. The command-queue task should be created with high priority so that it processes the command buffers in the queue timely and efficiently. A preprocessor symbol is defined for the priority value that user can modify for his application.

```
#ifndef QUEUE_TASK_PRIO
#define QUEUE_TASK_PRIO  (configMAX_PRIORITIES - 1)
#endif /* QUEUE_TASK_PRIO */
```

The GPU generates an interrupt after the commands in a command buffer are executed by hardware. Then the ISR of GPU sends a notification to command-queue task to notify it that the hardware completes all commands in the command buffer that the command-queue task just sent into hardware to execute.

Figure 3 shows the process of command buffer management in the VGLite driver. In some conditions, the driver must wait for the completion of the command buffer that is submitted just now. The driver calls `stall` method to wait for the completion of the commands in the command buffer.

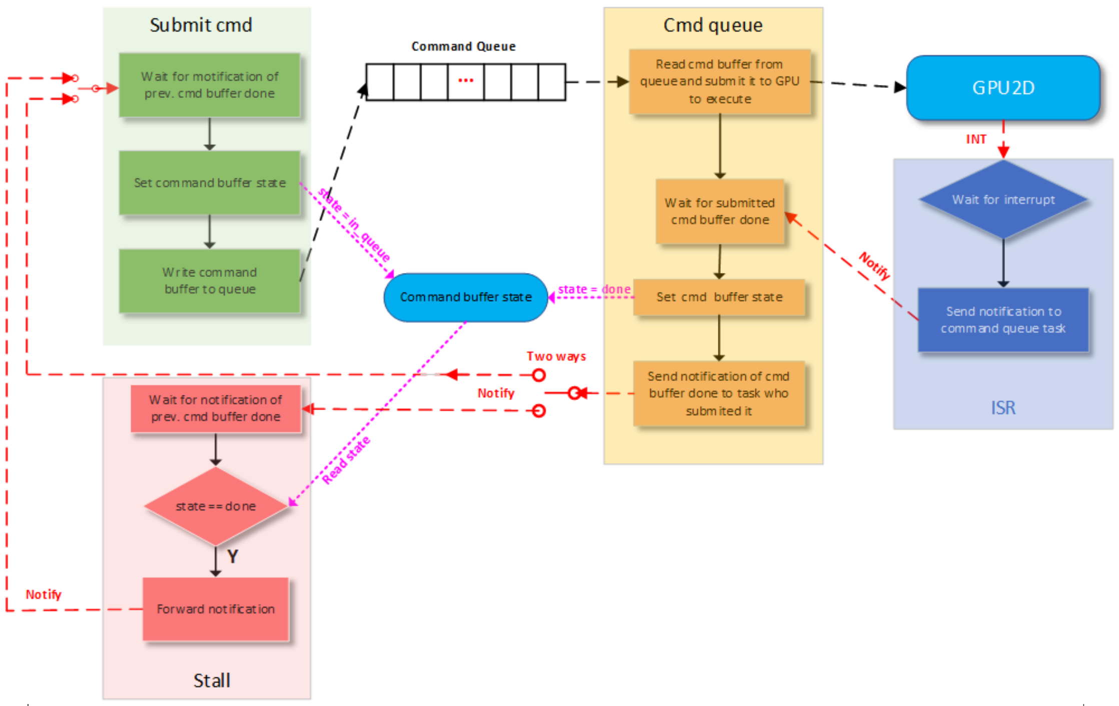
Submit cmd
Wait for notification of prev. cmd buffer done
Set command buffer state
Write command buffer to queue
Command Queue
...
Cmd queue
Read cmd buffer from queue and submit it to GPU to execute
Wait for submitted cmd buffer done
Set cmd buffer state
Send notification of cmd buffer done to task who submited it
GPU2D
INT
Wait for interrupt
Send notification to command queue task
ISR
Notify
state = in_queue
Command buffer state
state = done
Two ways
Notify
Wait for notification of prev. cmd buffer done
state == done
Y
Forward notification
Read state
Notify
Stall

Figure 3. Process of command buffer management

4 Porting driver for bare metal

The VGLite driver API in the SDK is implemented for FreeRTOS and support multiple tasks. Bare metal driver has no request of supporting multiple tasks. To use GPU hardware resource, call its APIs in sequence .

According to the mentioned code structure, directly reuse the code of OS independent layer directly since they are just logic code of API implementation without accessing hardware resource, or only accessing the code of OS abstract layer. To eliminate the effort of code maintenance and upgradation in the future, the porting reuses the code and changes the OS independent layer as little as possible.

The following subsections simplify the command buffer management for bare metal and describe the porting for different parts of VGLite middleware respectively.

4.1 Command buffer management

The OS independent layer uses two command buffers in ping-pang mode. But bare metal driver only needs to process the command buffers from the bare metal task. The command queue is not needed that makes the command buffer management for bare metal simpler than multiple tasks.

The prepared command buffer is submitted to GPU hardware directly. Same as multiple-task, only one command buffer can be submitted at any time. It is not mandatory for driver to wait for the completion of submitted commands processed by GPU after it's submitted. But before submitting the next command buffer, code needs to check if the executing of previous submitted commands is completed. The command buffer can only be submitted after the previous submitted commands are processed by GPU. Bare metal has no synchronization mechanism available in an OS but polling. Figure 4 shows the flow of a command buffer.

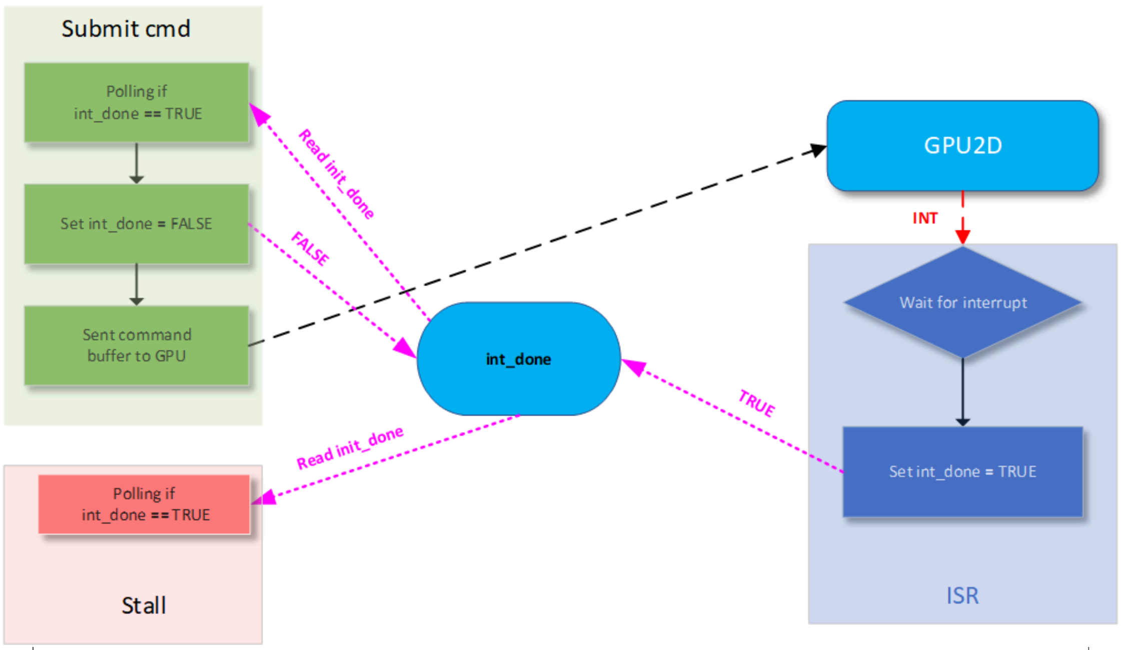

Figure 4. Command buffer flow

4.2 User API porting

4.2.1 OS independent layer

The code of OS independent layer can be reused. Due to command buffer management change, the below macro definition in the `vg_lite.c` should be changed.

```
#define CMDBUF_IN_QUEUE(context, id) \
        (vg_lite_os_event_state(&(context)->async_event[(id)])
 == VG_LITE_IN_QUEUE)
```

Considering the fact that a command buffer can only be submitted after the previous command buffer is completely processed, this check is actually redundant because the result is always false. The change is just to define it as FALSE.

```
#define CMDBUF_IN_QUEUE(context, id)  FALSE
```

To make driver efficient and code size small, the preprocessor directive can be added to exclude the related code snippet that checks this condition same as following:

```
#if !defined(_BAREMETAL)
    if(CMDBUF_IN_QUEUE(&context->context, command_id))
        VG_LITE_RETURN_ERROR(stall(context, 0));
#endif
```

All codes used for multitasks support can be isolated by preprocessor building variable. These codes include `flush()`, `update_context_buffer()`, `push_states_to_context()` and `has_valid_context_buffer()` functions in `vg_lite.c` file. They are never used in the bare metal case.

A new API `vg_lite_error_t vg_lite_query_idle(uint32_t* state)` is added to give application more flexibility to use VGLite. For example, application can call `vg_lite_flush` to replace `vg_lite_finish` to submit the commands to draw vector path or render image and not wait the commands done but do something else. Application does not check GPU status until it uses or renders the result of drawing.

4.2.2 OS-specific layer

The bare metal can be regarded as a special **OS** platform. The main modification of the porting for bare metal is to implement the bare metal specific code.

Based on the description of command buffer's flow afore, the command-buffer-queue is removed. The synchronization objects between command buffer queue and ISR, queue and task are not needed anymore.

Task local storage is implemented by a pointer variable as follows:

```
static void* pTLS;
int32_t vg_lite_os_set_tls(void* tls)
{
    if(tls == NULL)
        return VG_LITE_INVALID_ARGUMENT;
    pTLS = tls;
    return VG_LITE_SUCCESS;
}
void * vg_lite_os_get_tls( )
{
    return pTLS;
}
void vg_lite_os_reset_tls()
{
    pTLS = (void*)0;
}
```

The following functions of OS specific must be implemented as follows:

```
void * vg_lite_os_malloc(uint32_t size)
{
    return malloc(size);
}
void vg_lite_os_free(void * memory)
{
    free(memory);
}
void vg_lite_os_sleep(uint32_t msec)
{
    SDK_DelayAtLeastUs(msec * 1000,
 SDK_DEVICE_MAXIMUM_CPU_CLOCK_FREQUENCY);
}
int32_t vg_lite_os_initialize(void)
{
    int_done = TRUE;
    return VG_LITE_SUCCESS;
}
void vg_lite_os_deinitialize(void)
```

```
{
}
int32_t vg_lite_os_lock()
{
    return VG_LITE_SUCCESS;
}
int32_t vg_lite_os_unlock()
{
    return VG_LITE_SUCCESS;
}
```

The above functions can be defined as symbols to get small code size. For example, to define **vg_lite_os_malloc** as followings:

```
#define vg_lite_os_malloc(size) malloc(size)
```

The command buffer is submitted into GPU directly. A flag is used to indicate the status of the submitted command buffer. The flag is checked to get the GPU idle or busy.

```
static volatile uint32_t int_done;
int32_t vg_lite_os_submit(uint32_t context, uint32_t physical,
 uint32_t offset, uint32_t size, vg_lite_os_async_event_t
 *event)
{
    curContext = context;
    /* Wait previous command done */
    while(!int_done);
     /* Clear interrupt done flag */
    int_done = FALSE;
    vg_lite_hal_poke(VG_LITE_HW_CMDBUF_ADDRESS, physical +
 offset);
    vg_lite_hal_poke(VG_LITE_HW_CMDBUF_SIZE, (size +7)/8 );
    return VG_LITE_SUCCESS;
}
int32_t vg_lite_os_wait(uint32_t timeout,
 vg_lite_os_async_event_t *event)
{
    while(!int_done);
    return VG_LITE_SUCCESS;
}
void vg_lite_os_IRQHandler(void)
{
    uint32_t flags = vg_lite_hal_peek(VG_LITE_INTR_STATUS);
    if (flags) {
        /* Set interrupt done flags. */
        int_done = TRUE;
        if (IS_AXI_BUS_ERR(flags))
        {
            vg_lite_bus_error_handler();
        }
    }
}
```

4.3 Kernel driver

Bare metal, as a special **OS** platform, considering the VGLite driver architecture discussed above, is same as FreeRTOS to reuse the OS-specific code in the user API layer for the **kernel** layer. There is no concept of user space and kernel space.

As a result, the porting for kernel driver is trivial. It is almost to do nothing but just to remove the `vg_lite_hal_wait_interrupt` from the `vg_lite_hal.c` file. The bare metal driver uses the polling GPU busy statue to replace the notification mechanism of OS environment.

5 Porting driver to support single task

The VGLite driver in the SDK release package supports the multiple tasks. To support multiple tasks, the VGLite driver must submit the command buffers existing in the command queue into GPU to execute and get the notification of the command processed by hardware. Especially, save the hardware context of currently being processed command of a task correctly while the hardware switches to process the next command from the other task. Likewise, reload the previous saved context of the new task into the GPU before executing its command. To have maximum performance in the multitask case, the VGLite always appends an operation into the command buffer to flush the command execution result from internal cache into the buffer. These code and operations take time to be executed even if there’s only one task running in the system.

There is always only one task using VGLite driver in most embedded application. But the code and operations in the VGLite driver to support multiple tasks still need time to run. It is meaningful to modify the VGLite driver to only support single task to get a good performance.

The driver with RTOS but to support single task is very similar to the bare metal except that the synchronization object is available in RTOS. After submitting a command buffer into GPU hardware in the bare metal driver, replace the polling method with the synchronization mechanism provided by RTOS service. It is easy to use bare metal driver as the start of porting for single task driver with RTOS. Comparing with the bare metal driver, the porting for VGLite driver for single task only exists in the OS-specific layer. The OS independent layers are almost same as that of bare metal driver and can be reused. Figure 5 shows the working flow of single task driver with RTOS support.

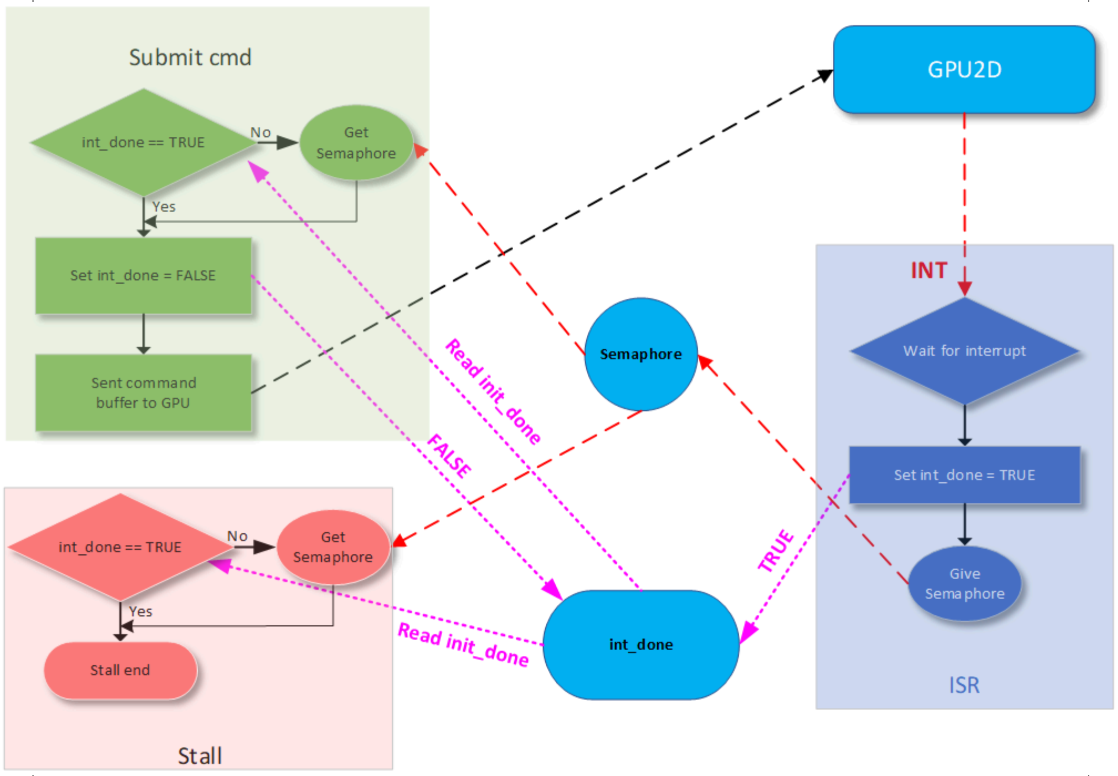

Figure 5. Working flow of single task driver with RTOS support

5.1 OS-specific layer

The porting OS-specific layer from the code of bare metal to support single task includes the following APIs:

- Map memory allocate/free APIs to memory allocation/free routines of RTOS. The below code is the example poring for FreeRTOS.

```
void * vg_lite_os_malloc(uint32_t size)
{
    return pvPortMalloc(size);
}

void vg_lite_os_free(void * memory)
{
    vPortFree(memory);
}
```

- Implement sleep API using RTOS's delay routine. It is same as the implementation for multiple tasks.

```
void vg_lite_os_sleep(uint32_t msec)
{
    vTaskDelay((configTICK_RATE_HZ * msec + 999) / 1000);
}
```

- Define a global semaphore variable `int_queue`, create a semaphore synchronization object in `vg_lite_os_initialize()`, and assign it to `int_queue`.

```
int32_t vg_lite_os_initialize(void)
```

```
{
    int_done = TRUE;
    int_queue = xSemaphoreCreateBinary();
    return VG_LITE_SUCCESS;
}
```

- In `vg_lite_os_wait()`, check whether GPU is still processing the command (`int_done` is false). If busy, wait for semaphore to become available.

```
int32_t vg_lite_os_wait(uint32_t timeout,
 vg_lite_os_async_event_t *event)
{
    if (int_done || (xSemaphoreTake(int_queue, timeout /
 portTick_PERIOD_MS) == pdTURE) {
      if (IS_AXI_BUS_ERR(int_flags))  {
          vg_lite_bus_error_handler();
      }
      int_flag = 0;
    }
    return VG_LITE_SUCCESS;
}
```

- In `vg_lite_os_submit()`, check whether the GPU is busy with command (`int_done` is false). If busy, wait for semaphore to become available.
- In GPU ISR, use the semaphore to wake up any code that is waiting for semaphore available.

```
void vg_lite_os_IRQHandler(void)
{
    uint32_t flags = vg_lite_hal_peek(VG_LITE_INTR_STATUS);

    portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;
  if (flags) {
        /* Set interrupt done flags. */
        int_done = TRUE;
    int_flags |= flags;

    /* Wake up any waiters. */
        if (int_queue) {
            xSemaphoreGiveFromISR(int_queue,
 &xHigherPriorityTaskWoken);
            if (xHigherPriorityTaskWoken != pdFALSE )
 {
                portYIELD_FROM_ISR(xHigherPriorityTaskWoken);

            }
        }
    }
```

6 Run examples

There are two projects named `clock_freertos` and `tiger_freertos` in the i.MX RT1170 SDK package used as the reference of the examples of AN13778SW to test the bare metal or single task VGLite drivers. The examples in AN13778SW have the similar code framework and exactly same function as those in SDK package. All code is based on the i.MX RT1170 SDK 2.11.x.

Figure 6 shows the cascade structure of VGLite driver in AN13778SW. The kernel FreeRTOS-specific layer is shared by multitask and single task.

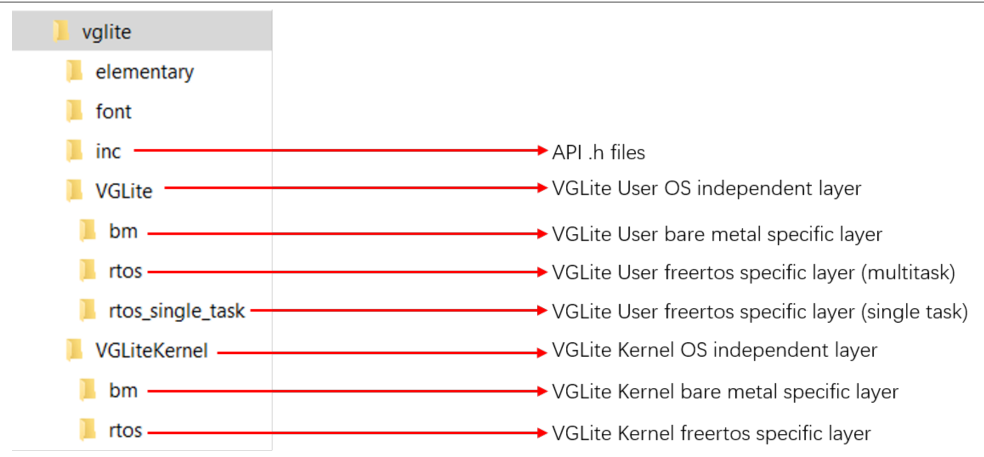
vglite
elementary
font
inc
VGLite
bm
rtos
rtos_single_task
VGLiteKernel
bm
rtos
API .h files
VGLite User OS independent layer
VGLite User bare metal specific layer
VGLite User freertos specific layer (multitask)
VGLite User freertos specific layer (single task)
VGLite Kernel OS independent layer
VGLite Kernel bare metal specific layer
VGLite Kernel freertos specific layer

Figure 6. VGLite driver

6.1 Baremetal examples

Two projects `clock_bm` and `tiger_bm` in AN13778SW are used to demonstrate the bare metal VGLite driver. To use the bare metal driver, the bare metal specific code under `VGLite/bm and VGLiteKernel/bm` is added into the project as OS-specific layer code. To isolate the unused code, define the preprocessor variable `_BAREMETAL`.

The additional `vg_lite_query_idle()` API is provided for application to query the GPU status. It is very useful for application to call `vg_lite_flush()` to submit all commands into GPU to process but not waiting them done. Instead, application calls `vg_lite_query_idle()` to check if GPU finishes all submitted commands before using the drawing result of a frame. With this method, the CPU can run other code while GPU is busy with executing the command.

6.2 Single task examples

The change for VGLite driver to support single task only includes the OS-specific layer. The RTOS environment and APIs are still exactly same as that of multiple tasks. And multiple tasks environment is unknown to an application. An application uses the VGLite when it is using the hardware without any concern about the other task who is also using GPU hardware at the same time. So, the code of examples in the SDK can be reused without any change.

As discussed in the previous paragraph, only OS-specific code of user layer in the driver needs to be changed. The driver code under `VGLite/rtos_single_task` replaces the code under `VGLite/rtos` in the default project. To isolate all unused code, define the preprocessor variable **ONE_TASK_SUPPORT**.

7 Conclusion

The VGLite driver code is well layered and only OS-specific layers need to be changed for bare metal or any other RTOS. For RTOS whose running is not separated as user space and kernel space, the kernel OS-specific layer can reuse the code of user OS-specific code.

The user API code in the `vg_lite.c` that is not used by bare metal and single task are well isolated by the preprocessor variable in AN13778SW. The future porting on the latest driver still can reference it. Because the isolated code is almost related to the multitask support.

According to our sanity test and customer's test, the driver only supporting single task has overall about 3 - 8 % performance improvement compared with multitask support.

8 References

1. *i.MX RT VGLite API Reference Manual* (document IMXRTVGLITEAPIRM)
2. *VGLite Driver Porting Guide* (document IMXRTVGLITEPG)

9 Revision history

Revision number	Date	Substantive changes
0	18 November 2022	Initial release

10 Legal information

10.1 Definitions

10.2 Disclaimers

10.3 Trademarks

Contents

Date of release: 18 November 2022
Document identifier: AN13778