NXP Semiconductors
User Guide

Document identifier: IMXRTVGLITEPG
Rev. 0, 11 February 2021

VGLite Driver Porting Guide

Contents

Chapter 1
Overview

This document outlines the key tasks required for porting the VGLite driver to a specific OS platform. With the provided reference implementation for FreeRTOS, it is straight forward to port the VGLite driver to other OS platforms. Users need to implement the HAL porting layer and the OS Abstraction layer (OSA) to make VGLite work on a specific OS platform.

VGLite software includes a lightweight API capable of running on various Vivante GC hardware.

NOTE

This document contains copyright material disclosed with permission of VeriSilicon Microelectronics.

NOTE

The document is intended for use with Vivante GC Cores and is compatible with software release versions starting from 3.0.0.

1.1 Requirements/Dependencies

Linux, FreeRTOS, Windows, or other OS-less environment.

VGLite software revision: 3.0.0, releases from November 2019 or later.

1.2 Software architecture

The VGLite API driver is organized into the following components:

- User API functions
- Kernel driver
- Kernel HAL layer
- Kernel OSA Layer

1.3 VGLite driver folder organization

`<Root_Dir>/inc/...` VGLite header files

`<Root_Dir>/VGLite/...` VGLite API functions

`<Root_Dir>/VGLite/rtos/...` VGLite FreeRTOS specific OSA implementation

`<Root_Dir>/VGLiteKernel/...` VGLite Kernel driver

`<Root_Dir>/VGLiteKernel/rtos/...` VGLite FreeRTOS platform-specific HAL implementation

1.3.1 User API

The User APIs and related enumerations are defined in `<Root_Dir>/inc/vg_lite.h`. The application must include this file for development. Refer to the Vivante VGLite Graphics API reference manual for more details on the APIs.

The User API functions implement the VGLite APIs as specified in the VGLite Graphics API reference manual. The application calls VGLite APIs. The User API functions call the underlying kernel functions to drive the GPU hardware.

Refer to subdirectory `<Root_Dir>/VGLite` for User API functions.

1.3.2 Kernel driver

The VGLite Kernel driver receives calls from User API functions and communicates with the GPU hardware to perform VGLite API operations. In a VGLite Linux implementation, the Kernel driver accepts `ioctl` commands from the User API functions to perform different operations. In the FreeRTOS implementation, the “Kernel” driver accepts direct function calls to perform different operations.

NOTE

Kernel resource is shared (includes the hardware states), so it must be guaranteed that the kernel module and hardware are initialized only once. Any initialization calls after that must be ignored and counted. The same convention is also applied to destroy. The last destroy call eventually releases the kernel and hardware resource. All "destroy" calls before that are ignored and counted.

For Kernel Driver functions, refer to subdirectory `<Root_Dir>/VGLiteKernel`.

1.3.3 Kernel HAL driver

The Kernel HAL Layer abstracts some basic hardware operations for the Kernel driver to communicate with GCNanoLiteV hardware. The operations include initialization and de-initialization of hardware, memory allocation and free, register read/write, and CPU/GPU synchronization. The HAL layer is OS independent so it can be supported on different platforms.

For HAL Layer functions, refer to `<Root_Dir>/inc/vg_lite_hal.h`.

1.3.4 Kernel OSA layer

The OS abstraction layer provides support for several OS services required by the VGLite kernel, such as synchronization objects (for example, mutexes, semaphores), work queues, tasks/threads, events, and OS memory management. When an operating system is present, the OSA layer only abstracts (hides) calls to the existing OS services. For the situations when no operating system is present, the OSA layer must implement these services itself.

Chapter 2
Porting the VGLite driver

The VGLite driver includes the HAL and OSA layers implementations for FreeRTOS. To port the VGLite driver to a new OS platform, implement the Kernel HAL and OSA layers to enable VGLite to work on that specific platform.

In some RTOS systems, such as FreeRTOS, there is no concept of User mode and Kernel mode. In these environments, the VGLite API functions call the “Kernel” functions in the same memory address space directly.

For a HAL layer implementation for FreeRTOS, check subdirectory `<Root_Dir>/VGLiteKernel/rtos`.

The corresponding OSA layer for FreeRTOS is available in the `<Root_Dir>/VGLite/rtos` subdirectory.

2.1 VGLite HAL interface

This section is dedicated to the details of VGLite HAL interface.

2.1.1 Initialization

```
void vg_lite_hal_initialize
     void
);
```

This function is called by the VGLite kernel before the GCPU hardware is initialized to allow SOC to turn on the power or initialize the clocks. The implementation must make sure that on the exit of this function the power and clock to GPU hardware is turned on and stable.

2.1.2 Deinitialization

```
void vg_lite_hal_deinitialize(
      void
);
```

This function is called by the VGLite kernel after the GPU hardware is uninitialized by the VGLite kernel, to allow SOC to perform system power control. On exit of this function, it is normal to have the power/clock to GPU hardware turned off.

2.1.3 Allocate/Free contiguous video memory

```
vg_lite_error_t vg_lite_hal_allocate_contiguous(
      unsigned long       size,
      void **             logical,
      int32_t *           physical,
      void **             node
);
void vg_lite_hal_free_contiguous(
      void *              memory_handle
);
```

This pair of HAL functions is used to dynamically allocate/free a buffer of contiguous video memory from the platform’s memory system. Any memory buffer that GPU hardware operates on must be allocated as a contiguous memory buffer. The allocated memory buffer is referenced by the VGLite driver through an opaque handle, usually a pointer to an opaque structure. The platform porting layer can put the necessary information inside this structure.

2.1.4 Register access

```
uint32_t vg_lite_hal_peek(
       uint32_t           address
);
void vg_lite_hal_poke(
       uint32_t           address,
       uint32_t           data
);
```

This pair of HAL functions reads data from a GPU register and writes data to a GPU register. The Vivante GPU register memory space must be mapped into a unified address space in the implementation so it can be accessed like a generic memory space.

2.1.5 Map/Unmap memory

```
void* vg_lite_hal_map(
       unsigned long      size,
       void *             logical,
       uint32_t           physical,
       uint32_t *         gpu
);
void vg_lite_hal_unmap(
      void *              memory_handle
);
```

This pair of HAL functions is used to map/unmap a contiguous logical or physical system memory buffer into the GPU device memory space. Any system memory buffer, such as a frame buffer or some pre-allocated image or path data, must be mapped into the GPU address space and referenced by a memory handle. It allows the GPU hardware to access the memory buffer directly. Either a logical or a physical address of a system memory buffer can be passed into the `vg_lite_hal_map` function.

Not all OS platforms must implement these functions. For example, on some RTOS OS all the memory buffers are statically pre-allocated. In such cases, the map/unmap functions can be empty.

2.1.6 Wait interrupt

```
int32_t vg_lite_hal_wait_interrupt(
      uint32_t            timeout,
      uint32_t            mask,
      uint32_t *          value
);
```

This function waits until an interrupt from the GPU hardware has been received. This HAL API is synchronous, which means it does not return until GPU hardware generates an interrupt, and the interrupt has been received by the operating system.

The function should wait for the specified number of milliseconds for the interrupt to occur. If the interrupt does not occur in the specified timeout period, the function returns with a timeout error.

2.1.7 Memory barrier

```
void vg_lite_hal_barrier(
      void
);
```

Some OS systems require a memory barrier to make sure all store operations by the CPU have been handled. This is the wrapper function for a platform specific memory barrier function.

2.2 VGLite OSA interface

This section describes the VGLite OS abstraction layer interface.

2.2.1 Initialization

```
int32_t vg_lite_os_initialize(void);
void vg_lite_os_deinitialize(void);
```

This pair of functions is used to initialize / to clean up the resources required by the OSA layer. They are called by the HAL layer after the initialization or deinitialization of the GPU hardware respectively.

```
 void vg_lite_os_sleep(uint32_t msec);
```

This function is required to suspend the execution of the current task for a specified number of milliseconds. It is required by the implementation of the “HAL delay” service.

2.2.2 Allocate/Free heap memory

```
 void * vg_lite_os_malloc(uint32_t size);
```

This function provides support for the allocation of heap memory. Heap memory may be managed by the operating system. It is required by the implementation of the HAL memory allocator.

```
 void vg_lite_os_free(void * memory);
```

This function is called to release heap memory that was previously allocated using the `vg_lite_os_malloc` API. It is required by the implementation of the HAL memory allocator.

2.2.3 Exclusive access

The VGLite kernel is counting on a global mutex to implement critical sections. Because all the VGLite compatible GPUs are equipped with a single core, the critical sections are protecting code paragraphs that are working directly with the GPU hardware.

```
int32_t vg_lite_os_lock(void);
```

This function should lock the GPU resource for the exclusive use of the current task/thread. It is used by the VGLite kernel during the configuration of the various GPU features.

```
int32_t vg_lite_os_unlock(void);
```

This function should unlock the GPU resource so that other tasks/threads may access it.

2.2.4 Task local storage

Task local storage support is required for the management of multiple drawing contexts. Each task is allowed to have one single VGLite context at any given time, created/destroyed using the `vg_lite_init` / `vg_lite_close` APIs.

```
int32_t vg_lite_os_set_tls(void* tls);
```

This function should save a provided pointer into the current Task Local Storage so that the VGLite HAL and kernel can later recover it by calling `vg_lite_os_get_tls`.

```
void * vg_lite_os_get_tls(void);
```

This function should return the pointer saved into the current Task Local Storage by a previous call to `vg_lite_os_set_tls` or NULL if no TLS pointer was saved by the current task.

```
void vg_lite_os_reset_tls(void);
```

This function should reset the pointer saved into the current Task Local Storage to NULL.

2.2.5 Synchronization

The VGLite HAL and kernel require task synchronization objects to correctly communicate with the GPU hardware.

```
int32_t vg_lite_os_init_event(vg_lite_os_async_event_t    *event,
                              uint32_t             semaphore_id,
                              int32_t              state);
```

This function should perform the following actions:

- check the consistency of the provided `event`
- create the binary semaphore designated by the provided `semaphore_id`
- link the binary semaphore to the provided `event` object
- set the state of the event attribute to the provided `state`.

```
int32_t vg_lite_os_delete_event(vg_lite_os_async_event_t *event);
```

This function should recycle all the resources allocated for the specified event and reset its attributes.

```
 int32_t vg_lite_os_wait_event(vg_lite_os_async_event_t *event);
```

This function should try to take ownership of the semaphore associated with the provided event. The function is expected to suspend the current task / thread until the event becomes available.

```
  int32_t vg_lite_os_wait(uint32_t timeout, vg_lite_os_async_event_t *event);
```

This function is used to wait for the GPU hardware to signal that the work is done. For that purpose the function is expected to wait a specified amount of time to get access to a specified event's semaphore and then check whether the state of the event is set to `VG_LITE_HW_FINISHED`. In case of success the function is expected to return the value `VG_LITE_SUCCESS`. If the event could not be accessed in time or if its state shows that the GPU did not successfully complete its task, the function is expected to return the value `VG_LITE_TIMEOUT`.

```
 int32_t vg_lite_os_signal_event(vg_lite_os_async_event_t *event);
```

This function is expected to release the semaphore associated with a specified event so that the tasks / threads waiting for it (that is, blocked in a `vg_lite_os_wait_event()` or `vg_lite_wait()` call) can be woken up.

2.3 VGLite API and HAL header files

In the VGLite source code:

- The Vivante VGLite API is defined in the file: `<Root_Dir>/inc/vg_lite.h`
- The VGLite HAL interface is defined in the file: `<Root_Dir>/inc/vg_hal_lite.h`
- The VGLite OSA layer interface is defined in the file: `<Root_Dir>/VGLite/<os>/vg_lite_os.h`

2.4 References

For details on the VGLite API, refer to the VGLite API Reference Manual, which is available in the i.MXRT platforms documentation package.

Chapter 3
Revision history

Table 1. Revision history

Revision number	Date	Substantive Changes
1.00	11 February 2021	Initial version.

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

Limited warranty and liability — Information in this document is provided solely to enable system and software implementers to use NXP products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. NXP reserves the right to make changes without further notice to any products herein.

NXP makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does NXP assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in NXP data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including “typicals,” must be validated for each customer application by customer's technical experts. NXP does not convey any license under its patent rights nor the rights of others. NXP sells products pursuant to standard terms and conditions of sale, which can be found at the following address: nxp.com/SalesTermsandConditions.

Right to make changes - NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Security — Customer understands that all NXP products may be subject to unidentified or documented vulnerabilities. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer’s applications and products. Customer’s responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer’s applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately. Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP. NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

For more information, please visit: http://www.nxp.com
For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 11 February 2021
Document identifier: IMXRTVGLITEPG