GR5525 Display Refresh Module Guide

Version: 1.2

Release Date: 2024-09-02

Shenzhen Goodix Technology Co., Ltd.

Trademarks and Permissions

GOODiX and other Goodix trademarks are trademarks of Shenzhen Goodix Technology Co., Ltd. All other trademarks and trade names mentioned in this document are the property of their respective holders.

Disclaimer

Information contained in this document is intended for your convenience only and is subject to change without prior notice. It is your responsibility to ensure its application complies with technical specifications.

Shenzhen Goodix Technology Co., Ltd. (hereafter referred to as “Goodix”) makes no representation or guarantee for this information, express or implied, oral or written, statutory or otherwise, including but not limited to representation or guarantee for its application, quality, performance, merchantability or fitness for a particular purpose. Goodix shall assume no responsibility for this information and relevant consequences arising out of the use of such information.

Without written consent of Goodix, it is prohibited to use Goodix products as critical components in any life support system. Under the protection of Goodix intellectual property rights, no license may be transferred implicitly or by any other means.

Shenzhen Goodix Technology Co., Ltd.

Headquarters: Floor 13, Phase B, Tengfei Industrial Building, Futian Free Trade Zone, Shenzhen, China

TEL: +86-755-33338828 Zip Code: 518000

Website: www.goodix.com

Preface

Purpose

This document introduces the display refresh module in GR5525 System-on-Chips (SoCs) which is applied to wearable devices, to help users quickly get started with the functionalities and features of the module and accelerate development and performance optimization of wearable devices.

Audience

This document is intended for:

- Device user
- Developer
- Test engineer
- Hobbyist developer
- Technical writer

Release Notes

This document is the third release of *GR5525 Display Refresh Module Guide*, corresponding to GR5525 SoC series.

Revision History

Version	Date	Description
1.0	2023-08-30	Initial release
1.1	2024-04-02	Updated "Overview".
1.2	2024-09-02	Deleted the GR5525IGNI SoC.

Contents

1 Overview

The GR5525 series comes in two package choices: QFN56 and QFN68. The specific configurations are listed below.

Table 1-1 Configuration of GR5525 series

GR5525 Series	GR5525RGNI	GR5525IENI	GR5525I0NI
CPU	Cortex®-M4F	Cortex®-M4F	Cortex®-M4F
RAM	256 KB	256 KB	256 KB
SiP Flash	1 MB	512 KB	N/A
I/O Number	50	39	39
I/O Voltage	1.8 V–3.6 V	1.8 V–3.6 V	In line with Flash voltage
Package (mm)	QFN68 (7.0 x 7.0 x 0.85)	QFN56 (7.0 x 7.0 x 0.75)	QFN56 (7.0 x 7.0 x 0.75)

1.1 Display Refresh Elements

GR5525 provides rich sets of display refresh elements, as listed below.

Table 1-2 GR5525 display refresh element summary 1

System Main Frequency	X-Flash	External QSPI Frequency	SRAM	QSPI0	QSPI1
96 MHz	512 KB/1 MB or external Flash	48 MHz	256 KB	• NOR Flash (read in XIP mode) • NAND Flash (access in register mode) • Q-PSRAM (read/write in XIP mode) • Display • DMA0 (register mode) • DMA0/DMA1 (XIP mode) • Dynamic adaptive endian • Multiple types of static endian • 1-line/2-line/4-line mode	• NOR Flash (read in XIP mode) • NAND Flash (access in register mode) • Q-PSRAM (read/write in XIP mode) • Display • DMA0/DMA1 (register mode) • DMA0/DMA1 (XIP mode) • Dynamic adaptive endian • Multiple types of static endian • 1-line/2-line/4-line mode

Table 1-3 GR5525 display refresh element summary 2

QSPI2	DMA	SPIM	DSPI
• NOR Flash (read in XIP mode) • NAND Flash (access in register mode) • Q-PSRAM (read/write in XIP mode) • Display • DMA1 (register mode)	• 2 instances • Linked List • Scatter • Gather • Large FIFO depth	48 MHz	48 MHz

QSPI2	DMA	SPIM	DSPI
• DMA0/DMA1 (XIP mode) • Dynamic adaptive endian • Multiple types of static endian • 1-line/2-line/4-line mode			

Major display refresh elements are categorized as follows:

1. Computing power
   (1) CPU
2. Persistent storage of materials/texture data
   (1) NOR Flash
   (2) NAND Flash
   (3) XQSPI Flash (remaining code space, part of which can be used to store data)
3. Computation and buffer space
   (1) 256 KB on-chip SRAM
   (2) External QSPI PSRAM up to 512 Mbit
4. Data transmission enhancement/Frame rate acceleration
   (1) DMA0/DMA1
5. Display interfaces (covering MIPI DBI Type-C interface)
   (1) QSPI interface
   (2) SPI master interface
   (3) Display SPI interface (3-line/4-line SPI)

1.2 Display Refresh Models

According to the market positioning of different wearable products, multitude of display refresh models can be built with various combinations of display refresh elements.

This document focuses on typical features and applications of modules such as DMA and QSPI. For introduction to each module, refer to *GR5525 Datasheet*.

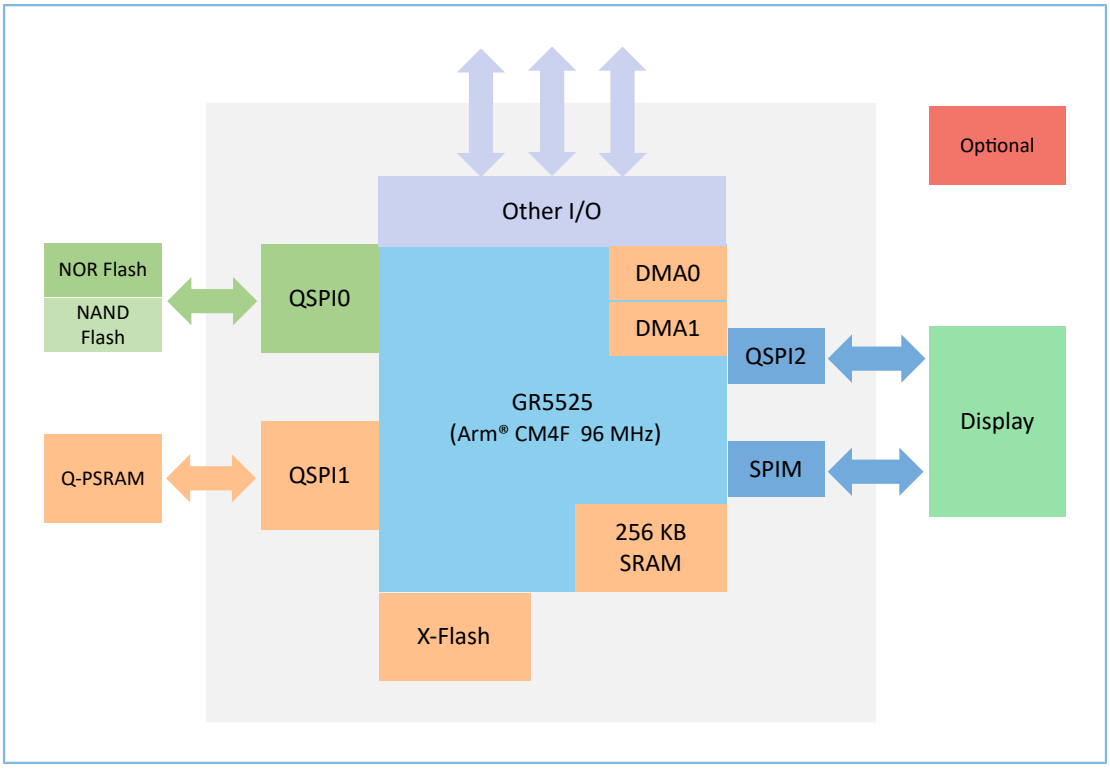


Figure 1-1 Typical display refresh model diagram of GR5525

2 Typical DMA Features

For details of Direct Memory Access (DMA), refer to DMA-related contents in *GR5525 Datasheet*.

2.1 Typical DMA Applications

2.1.1 DMA Base Data Transmission

DMA enables GR5525 to transmit data from memory to memory, memory to peripheral, peripheral to memory, and peripheral to peripheral.

In a DMA base data transmission, up to 4095 beats of data can be transmitted at a time. “Beat” refers to the bit width of the data transmitted through DMA.

- If the data bit width is in bytes, up to 4095 bytes of data can be transmitted in a single DMA transmission.
- If the data bit width is in half-words, up to 4095 x 2 bytes of data can be transmitted in a single DMA transmission.
- If the data bit width is in words, up to 4095 x 4 bytes of data can be transmitted in a single DMA transmission.

If more than 4095 beats of data are transmitted in a single DMA base data transmission, an error message will pop up, indicating that the DMA transmission is interrupted.

Note:

Address alignment shall be ensured in DMA transmission.

2.1.2 DMA Chain Data Transmission

To improve the DMA transmission capability in the following scenarios, GR5525 SoCs introduce DMA chain data transmission, which means multiple data blocks can be connected using a pointer linked list, so that all these data can be transmitted in a single DMA transmission cycle.

- Transmit more than 4095 beats of data in a single transmission.
- Transmit data from a discontinuous address space in a single transmission cycle.
- Use different transmission configurations to transmit data in a single transmission cycle.

To be specific: Manage all to-be-transmitted data blocks using a pointer linked list. After a data block is sent, the next block is automatically loaded according to the .next pointer until the .next pointer becomes empty.

As shown below, each link node mainly contains DMA transmission configuration information of the current node, node data block, and information of the pointer pointing to the next link node, until the last transmission link node becomes null. These link nodes will be implemented in a single DMA transmission. Data at each node shall not be more than 4095 beats.

DMA chain data transmission can be widely used in such scenarios as big data block transmission, discontinuous data transmission, and display refresh.

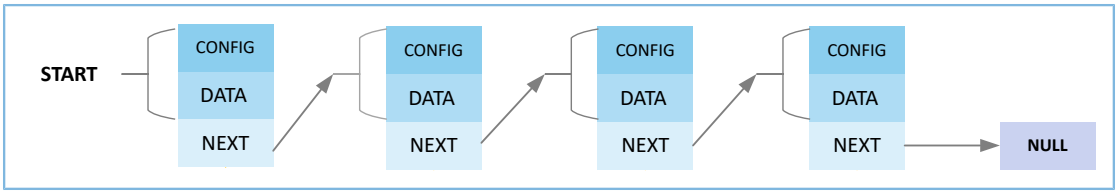


Figure 2-1 DMA chain data transmission

2.1.3 DMA Scatter Transmission

In a DMA scatter transmission, data in a continuous address space is scattered to a discontinuous address space based on certain rules.

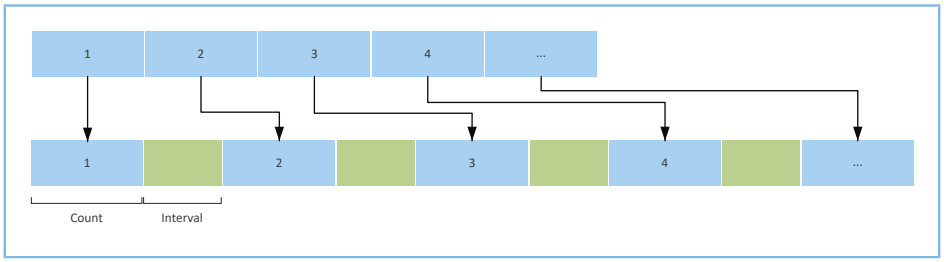


Figure 2-2 DMA scatter transmission schematic

1. The continuous data is divided equally into several data blocks, and the length of the last data block can be different.
2. The data volume contained in each data block is marked as “count” (in beats).
3. The address interval between data blocks is marked as “interval” (in beats).

Example: To transmit 1000 bytes of data (count: 4; interval: 2) through DMA scatter transmission:

- If the transmission width is 8 bits, then the data is 1000 beats in total, each data block is 4 bytes, and the address interval is 2 bytes.
- If the transmission width is 16 bits, then the data is 500 beats in total, each data block is 8 bytes, and the address interval is 4 bytes.
- If the transmission width is 32 bits, then the data is 200 beats in total, each data block is 16 bytes, and the address interval is 8 bytes.

2.1.4 DMA Gather Transmission

In a DMA gather transmission, data in a discontinuous address space is gathered in a continuous address space. DMA gather transmission can be regarded as a reverse process of DMA scatter transmission.

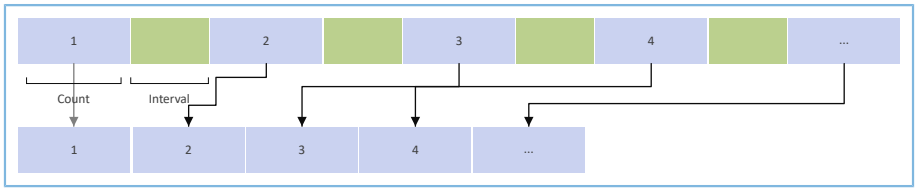


Figure 2-3 DMA gather transmission schematic

1. Several data blocks of equal length are evenly arranged at the same address interval, and the length of the last data block can be different.
2. The data volume contained in each data block is marked as “count” (in beats).
3. The address interval between data blocks is marked as “interval” (in beats).

Example: To transmit 1000 bytes of data (count: 4; interval: 2) through DMA gather transmission:

- If the transmission width is 8 bits, then the data is 1000 beats in total, each data block is 4 bytes, and the address interval is 2 bytes.
- If the transmission width is 16 bits, then the data is 500 beats in total, each data block is 8 bytes, and the address interval is 4 bytes.
- If the transmission width is 32 bits, then the data is 200 beats in total, each data block is 16 bytes, and the address interval is 8 bytes.

2.1.5 DMA Scatter and Gather Transmission

The DMA scatter and gather transmission can be enabled simultaneously, to transmit the data in a discontinuous address space to another discontinuous address space.

2.2 Features of DMA Transmission Channel

GR5525 provides two DMA instances for users: DMA0 and DMA1. There are six DMA channels in total. The FIFO depth is 32 (channel 0) or 4 (channels 1–5).

Channel 0 features large FIFO depth, so it can cache more data in DMA transmission, improving DMA transmission throughput. Therefore, in wearable applications, it is recommended to allocate channel 0 of DMA0/DMA1 to peripherals with high throughput in such scenarios as Flash access, PSRAM access, bulk memory block transfer, and display.

3 Typical QSPI Features

For details of QSPI, refer to QSPI-related contents in *GR5525 Datasheet*.

3.1 QSPI Working Modes

GR5525 QSPI works in three modes according to the number of data cables in use and the sequence:

- 1-line SPI mode: SCLK, CS#, MOSI, and MISO signals are involved.
- 2-line Dual SPI mode: SCLK, CS#, IO0, and IO1 signals are involved.
- 4-line Quad SPI mode: SCLK, CS#, IO0, IO1, IO2, and IO3 signals are involved.

GR5525 QSPI works in two modes according to the specific peripheral access methods:

- Register mode: After initialization of the QSPI module and peripherals, QSPI reads data from/writes data to peripherals by controlling registers. These control operations are encapsulated as functions in SDK by functionalities, and specific accesses are implemented by calling corresponding functions (arrays).
- Memory mapped mode: also called “XIP mode”. After initialization of the QSPI module and peripherals, the memory space of peripherals is mapped to the bus address space of the system, and then QSPI accesses peripherals by bus addressing.

Generally, read/write access to any peripheral that supports the QSPI sequence protocol can be implemented in register mode, such as NOR Flash, NAND Flash, PSRAM, display, and other peripherals with QSPI interface and supporting QSPI sequence.

QSPI NOR Flash and QSPI PSRAM can also be accessed in memory mapped mode. Due to different access characteristics, QSPI NOR Flash only supports read operations in memory mapped mode, whereas QSPI PSRAM supports both read and write operations in this mode.

Note:

Generally, storage devices supporting memory mapped mode can only work in Dual SPI/Quad SPI mode. Additionally, to ensure higher access efficiency, it is recommended to work in Quad SPI mode.

The table below lists the access methods supported by common wearable peripherals with recommendations as follows:

1. For specific modes supported by a peripheral in practice, you can refer to the datasheet of the peripheral.
2. It is recommended to adopt the memory mapped mode if supported by peripherals, to make the program code used to read data from the storage device more concise and access more efficient.

Table 3-1 GR5525 QSPI working modes supported by wearable peripherals

Working Mode		NOR Flash		NAND Flash		QSPI PSRAM		Display/LCD	
		Read	Write	Read	Write	Read	Write	Read	Write
Register Mode	SPI	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
	Dual SPI	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Working Mode		NOR Flash		NAND Flash		QSPI PSRAM		Display/LCD	
		Read	Write	Read	Write	Read	Write	Read	Write
	Quad SPI	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Memory Mapped Mode	SPI	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
	Dual SPI	Yes	N/A	N/A	N/A	Yes	Yes	N/A	N/A
	Quad SPI	Yes	N/A	N/A	N/A	Yes	Yes	N/A	N/A

3.2 QSPI Data Endian

3.2.1 Data Endian in Write Operations

When the same memory data is written to a peripheral through QSPI with different bus access widths using CPU or DMA, different types of byte endian will be obtained, as shown below.

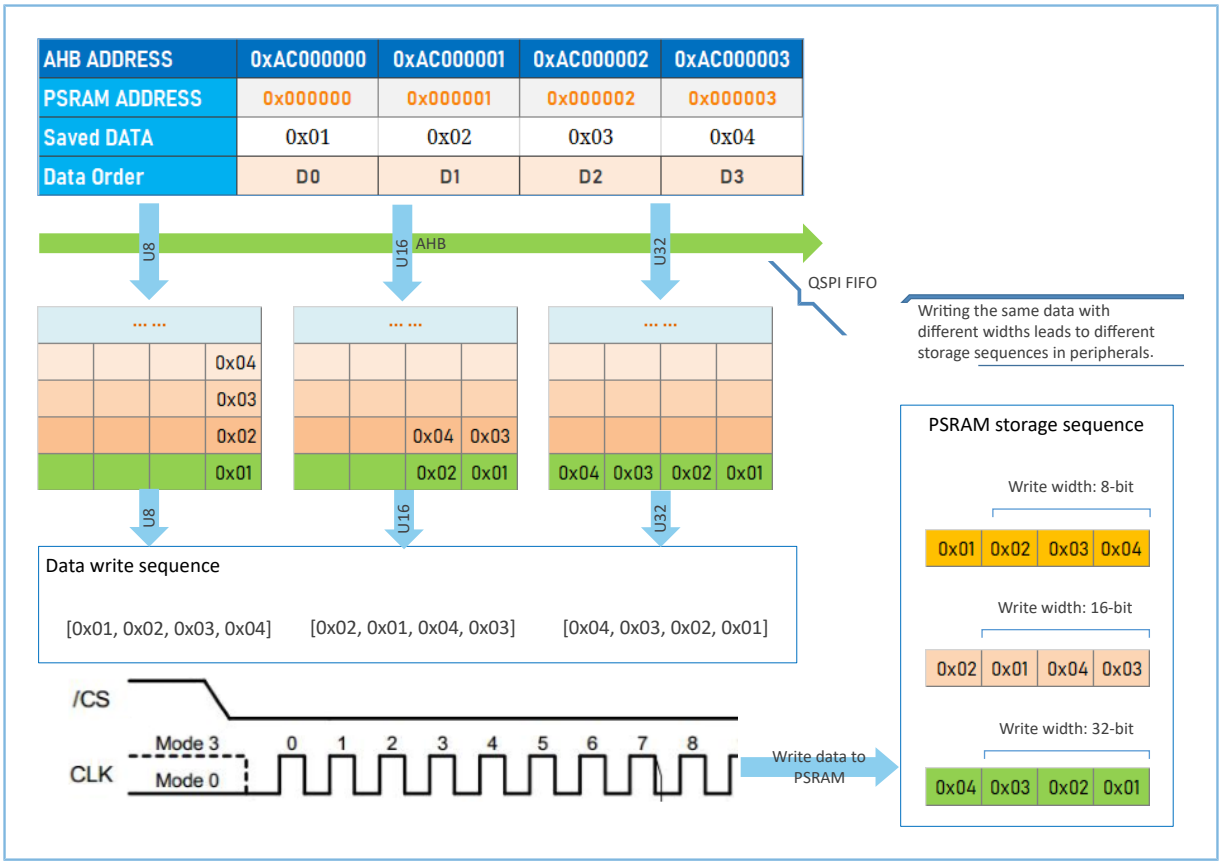


Figure 3-1 Default write process of Synopsys QSPI controller

The endian conversion process is as follows:

1. Send the array {0x01, 0x02, 0x03, 0x04} through QSPI.
2. When the data enters the QSPI FIFO queue:

(1) If the data is to be written by bytes, 4 FIFO depths will be occupied. The array is arranged as 0x01, 0x02, 0x03, 0x04 in order.

(2) If the data is to be written by half-words, the data is converted into 0x0201 and 0x0403 on the bus, and 2 FIFO depths will be occupied in QSPI. The array is arranged as {0x02, 0x01} and {0x04, 0x03} in order.

(3) If the data is to be written by words, the data is converted into 0x04030201 on the bus, and 1 FIFO depth will be occupied in QSPI. The array is arranged as {0x04, 0x03, 0x02, 0x01}.

3. QSPI FIFO outputs data through data cables from MSB to LSB, so the order of the data in data cables is shown below:

(1) If the data is to be written by bytes, the data in the sequence line is displayed as 0x01, 0x02, 0x03, 0x04.

(2) If the data is to be written by half-words, the data in the sequence line is displayed as 0x02, 0x01, 0x04, 0x03.

(3) If the data is to be written by words, the data in the sequence line is displayed as 0x04, 0x03, 0x02, 0x01.

4. Peripherals store/process the data according to the data endian.

Note:

By default, data write operations in both register mode and memory mapped mode follow the above endian conversion rules.

3.2.2 Data Endian in Read Operations

Read operations are a reverse process when compared with write operations. Different read access widths are adopted for the data in the address space of peripherals, to obtain data with different types of byte endian, as shown below.

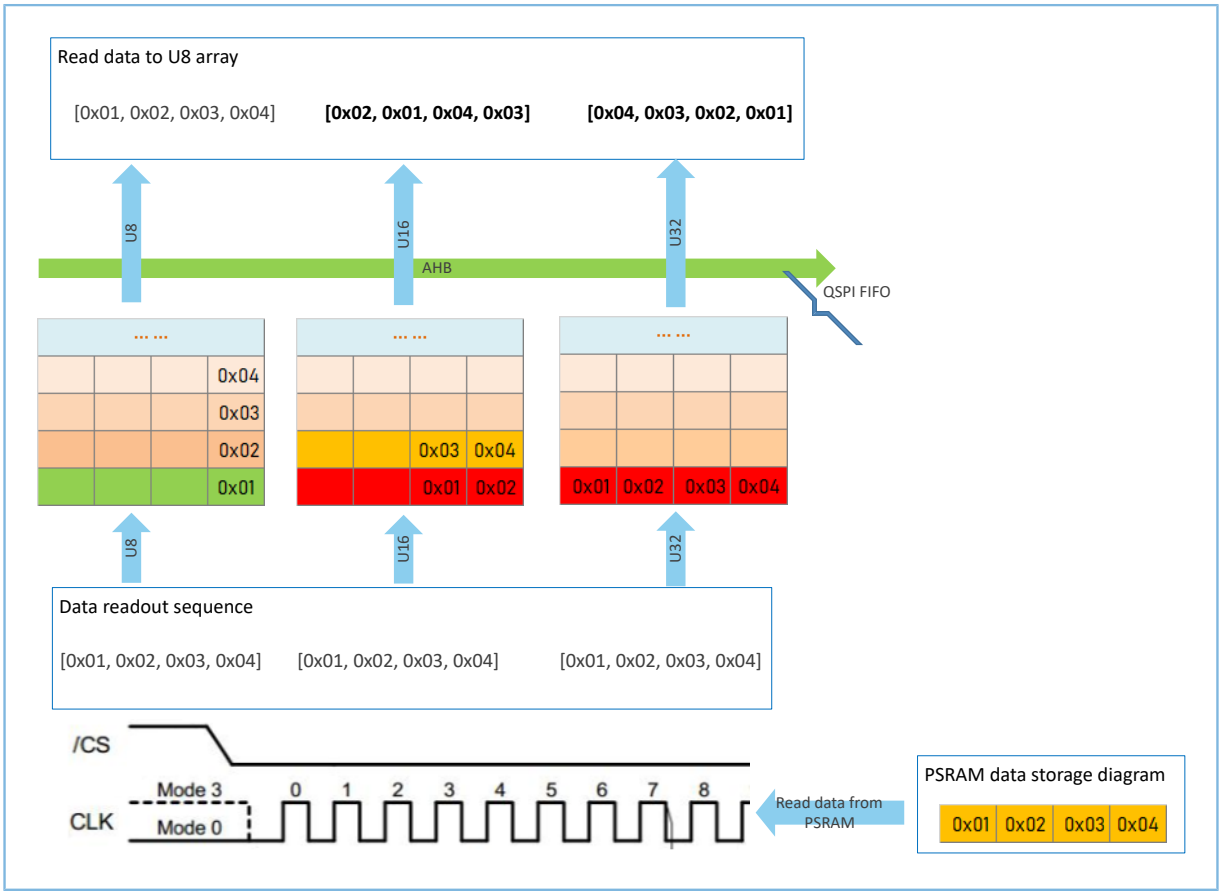


Figure 3-2 Data endian in read operations

The endian conversion process is as follows:

- If the data width is in bytes, read {0x01, 0x02, 0x03, 0x04} in the peripheral space to the memory space as {0x01, 0x02, 0x03, 0x04}.
- If the data width is in half-words, read {0x01, 0x02, 0x03, 0x04} in the peripheral space to the memory space as {0x02, 0x01, 0x04, 0x03}.
- If the data width is in words, read {0x01, 0x02, 0x03, 0x04} in the peripheral space to the memory space as {0x04, 0x03, 0x02, 0x01}.

By default, data read operations in both register mode and memory mapped mode follow the above endian conversion rules. To obtain the same byte endian, access widths of read and write operations shall be kept the same.

However, in wearable product applications, the minimum access unit of resources such as images and fonts may be byte, half-word, word, or any combination of them due to different color formats and color depths. Therefore, processing of byte endian is required during software access. If endian is adjusted by software, a lot of CPU computing power will be consumed. GR5525 SoCs are designed with endian modes supporting various access scenarios.

3.2.3 Read Rule for Static Data Endian in Memory Mapped Mode

The static data endian refers to a data endian in which data is output correspondingly according to different register configurations (hereinafter referred to as “static endian rule”).

- The static endian rule applies only when QSPI works in memory mapped mode.
- The static endian rule is used to read data from QSPI NOR Flash.

According to the static endian rule, for a fixed data storage sequence, you can configure registers to output different types of data read endian. The table below lists data stored in peripheral memory.

Table 3-2 Peripheral data to be read

Peripheral Address	0x000000	0x000001	0x000002	0x000003
Stored Data	0x01	0x02	0x03	0x04

Store the data 0x01, 0x02, 0x03, and 0x04 in the peripheral address space 0x000000–0x000003 in order.

The table below lists the data obtained when different data types are adopted to access the above peripheral data with different endian rules.

Table 3-3 Static endian rule for QSPI read operations

Access Type	Byte				Half-word		Word
Access Address	0xAC000000	0xAC000001	0xAC000002	0xAC000003	0xAC000000	0xAC000002	0xAC000000
Endian Mode 0	0x01	0x02	0x03	0x04	0x0102	0x0304	0x01020304
Endian Mode 1	0x01	0x02	0x03	0x04	0x0201	0x0403	0x02010403
Endian Mode 2	0x01	0x02	0x03	0x04	0x0201	0x0403	0x04030201

Descriptions about the static endian rule:

- Assume that the mapping address of the peripheral address 0x000000–0x000003 in the bus address space is 0xAC000000–0xAC000003.
- When static endian mode 0 is configured:
  - When 0xAC000000 is accessed in bytes (corresponding to uint8_t * in C language), 0x01 is returned.
  - When 0xAC000000 is accessed in half-words (corresponding to uint16_t * in C language), 0x0102 is returned.
  - When 0xAC000000 is accessed in words (corresponding to uint32_t * in C language), 0x01020304 is returned.
- When static endian mode 1 is configured:
  - When 0xAC000000 is accessed in bytes (corresponding to uint8_t * in C language), 0x01 is returned.
  - When 0xAC000000 is accessed in half-words (corresponding to uint16_t * in C language), 0x0201 is returned.
  - When 0xAC000000 is accessed in words (corresponding to uint32_t * in C language), 0x02010403 is returned.
- When static endian mode 2 is configured:
  - When 0xAC000000 is accessed in bytes (corresponding to uint8_t * in C language), 0x01 is returned.

- When 0xAC000000 is accessed in half-words (corresponding to uint16_t * in C language), 0x0201 is returned.
- When 0xAC000000 is accessed in words (corresponding to uint32_t * in C language), 0x04030201 is returned.

You can set a proper endian rule on demand to ensure optimal efficiency of QSPI.

Set the function for the static endian rule: app_qspi_mmap_set_endian_mode(). For details, refer to "Section 3.3.3.4 app_qspi_mmap_set_endian_mode".

3.2.4 Read Rule for Dynamic Data Endian in Memory Mapped Mode

The dynamic data endian refers to a data read endian in which data is output in memory mapped mode according to different data access types (hereinafter referred to as "dynamic endian rule").

- The dynamic endian rule applies only when QSPI works in memory mapped mode.
- The dynamic endian rule is typically used to read data from/write data to QSPI PSRAM.
- The dynamic endian rule has a higher priority than the static endian rule. If both dynamic and static endian rules are enabled, the system responds to the dynamic endian rule only.

According to the dynamic endian rule, when a read/write access occurs, the behavior of data entering FIFO can be modified to automatically adapt to mixed access to different types of data.

The figure below shows the behavior of different types of data entering QSPI FIFO during read and write operations after the dynamic endian rule is enabled. The dynamic endian rule helps ensure write consistency for different types of base data.

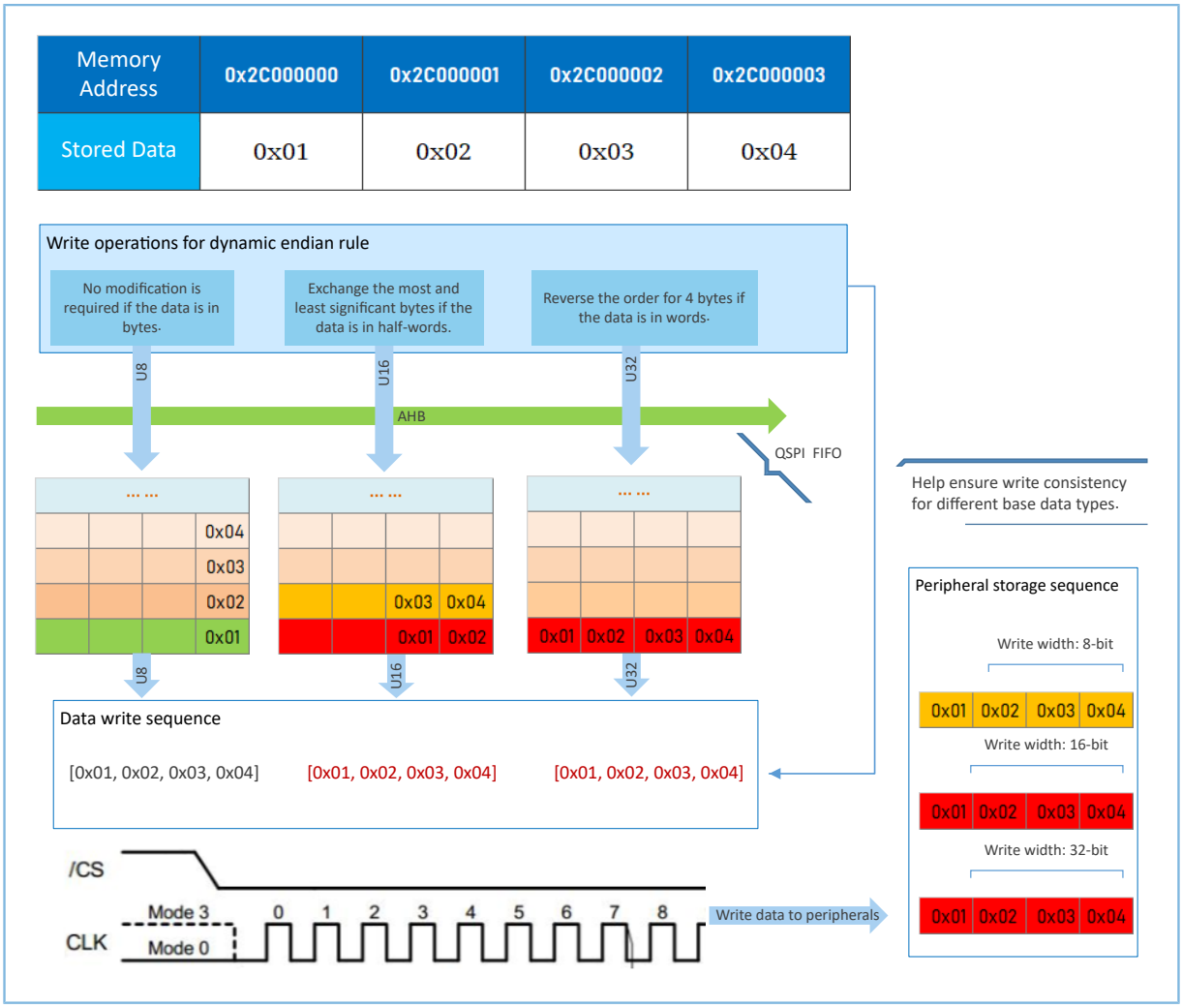


Figure 3-3 Behavior of different types of data entering QSPI FIFO for write operations

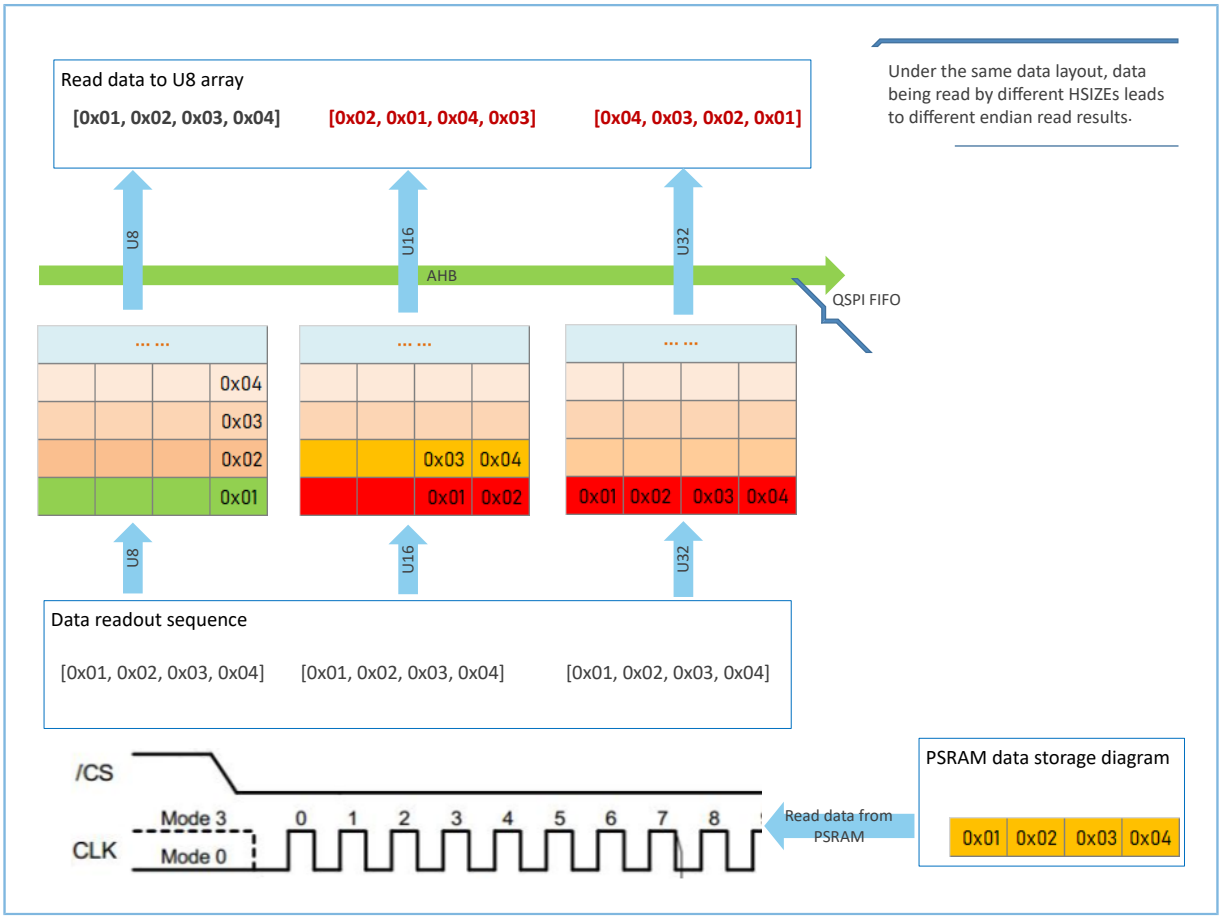


Figure 3-4 Behavior of different types of data entering QSPI FIFO for read operations

When QSPI PSRAM initializes the memory mapped mode, the GR5525 driver automatically enables the dynamic endian rule, to make PSRAM access consistent with SRAM access.

3.3 QSPI Driver Operational Instructions

3.3.1 Common Functions

3.3.1.1 app_qspi_init

Table 3-4 app_qspi_init API

Function Prototype	uint16_t app_qspi_init(app_qspi_params_t *p_params, app_qspi_evt_handler_t evt_handler)
Function Description	Initialize the QSPI module.
Parameter	• p_params: initialization parameters • evt_handler: Register the function for notifying asynchronous API callback events.
Return Value	Refer to APP_DRV_ERR_CODE in the source code.
Remarks	

app_qspi_params_t definition and members are detailed as follows.

```
typedef struct
{
    app_qspi_id_t          id;
    app_qspi_pin_cfg_t     pin_cfg;
    app_qspi_dma_cfg_t     dma_cfg;
    qspi_init_t            init;
} app_qspi_params_t;
```

id

- APP_QSPI_ID_0: QSPI0
- APP_QSPI_ID_1: QSPI1
- APP_QSPI_ID_2: QSPI2

pin_cfg

- cs: Configure the CS pin.
  - type: pin type
  - mux: Configure I/O multiplexing.
  - pin: pin number
  - mode: Configure the I/O mode.
  - pull: Activate pull-up or pull-down resistors.
  - enable: Enable/Disable the CS pin.
- clk: Configure the clock pin.
- io_0: QSPI IO0 (SPI MOSI)
- io_1: QSPI IO1 (SPI MISO)
- io_2: QSPI IO2
- io_3: QSPI IO3

By default, three groups of QSPI pin configurations are provided by the driver and are defined in g_qspi_pin_groups, which can be used directly or redefined as needed.

dma_cfg

- dma_instance: Allocate a DMA instance to the selected QSPI module (ID) when DMA transfer is adopted, with DMA0 for QSPI0, DMA0/DMA1 for QSPI1, and DMA2 for QSPI2.
- dma_channel: Allocate a channel to the DMA instance when DMA transfer is adopted.
- wait_timeout_ms: Set the timeout period (in ms) for polling APIs.
- extend: bits reserved for future use

init

- clock_prescaler: QSPI clock prescaler value, which uses the system peripheral clock as the baseline and can be an even number in the range of 2–65535
- clock_mode: clock mode. Four modes are available based on different clock edge and phase configurations.
- rx_sample_delay: delayed clock cycle(s) of RX sampling. You can set it to “1” for the maximum frequency and “0” for other frequencies.

3.3.1.2 app_qspi_deinit

Table 3-5 app_qspi_deinit API

Function Prototype	uint16_t app_qspi_deinit(app_qspi_id_t id)
Function Description	Deinitialize the QSPI module.
Parameter	id: QSPI module ID
Return Value	Refer to APP_DRV_ERR_CODE in the source code.
Remarks	

3.3.1.3 app_qspi_get_handle

Table 3-6 app_qspi_get_handle API

Function Prototype	qspi_handle_t *app_qspi_get_handle(app_qspi_id_t id)
Function Description	Obtain the QSPI control handle based on corresponding module ID.
Parameter	id: QSPI module ID
Return Value	Pointer to the QSPI control handle
Remarks	

3.3.1.4 app_qspi_dma_init

Table 3-7 app_qspi_dma_init API

Function Prototype	uint16_t app_qspi_dma_init(app_qspi_params_t *p_params)
Function Description	Initialize QSPI DMA mode.
Parameter	p_params: pointer to the initialization parameter structure
Return Value	APP_DRV_xxx: Refer to the macro definitions in SDK_Folder\drivers\inc\app_drv_error.h.
Remarks	

3.3.1.5 app_qspi_dma_deinit

Table 3-8 app_qspi_dma_deinit API

Function Prototype	uint16_t app_qspi_dma_deinit(app_qspi_id_t id)
Function Description	Deinitialize QSPI DMA mode.

Parameter	id: QSPI module ID
Return Value	APP_DRV_xxx: Refer to the macro definitions in SDK_Folder\drivers\inc\app_drv_error.h.
Remarks	

3.3.2 APIs in Register Mode

In register mode, these APIs serve as generic interfaces to access any QSPI peripheral.

- Based on application scenarios, they are categorized into two groups: APIs with command control and APIs without command control.
- Based on data transmission modes, they are categorized into three groups: app_xxx_transmit_sync, app_xxx_transmit_async, and app_xxx_dma_transmit_async, corresponding to data transmission in polling mode, interrupt mode, and DMA mode respectively.

3.3.2.1 app_qspi_command_sync

Table 3-9 app_qspi_command_sync API

Function Prototype	uint16_t app_qspi_command_sync(app_qspi_id_t id, app_qspi_command_t *p_cmd, uint32_t timeout)
Function Description	Transmit commands synchronously in QSPI polling mode.
Parameter	• id: QSPI module ID • p_cmd: buffer to store the command to be transmitted • timeout: timeout period (unit: ms)
Return Value	APP_DRV_xxx: Refer to the macro definitions in SDK_Folder\drivers\inc\app_drv_error.h.
Remarks	

3.3.2.2 app_qspi_command_async

Table 3-10 app_qspi_command_async API

Function Prototype	uint16_t app_qspi_command_async(app_qspi_id_t id, app_qspi_command_t *p_cmd)
Function Description	Transmit commands asynchronously in QSPI interrupt mode.
Parameter	• id: QSPI module ID • p_cmd: buffer to store the command to be transmitted
Return Value	APP_DRV_xxx: Refer to the macro definitions in SDK_Folder\drivers\inc\app_drv_error.h.
Remarks	

3.3.2.3 app_qspi_dma_command_async

Table 3-11 app_qspi_dma_command_async API

Function Prototype	uint16_t app_qspi_dma_command_async(app_qspi_id_t id, app_qspi_command_t *p_cmd)

Function Description	Transmit commands asynchronously in QSPI DMA mode.
Parameter	• id: QSPI module ID • p_cmd: buffer to store the command to be transmitted
Return Value	APP_DRV_xxx: Refer to the macro definitions in SDK_Folder\drivers\inc\app_drv_error.h.
Remarks	

3.3.2.4 app_qspi_command_receive_sync

Table 3-12 app_qspi_command_receive_sync API

Function Prototype	uint16_t app_qspi_command_receive_sync(app_qspi_id_t id, app_qspi_command_t *p_cmd, uint8_t *p_data, uint32_t timeout)
Function Description	Read data synchronously in QSPI polling mode, with a control command sent first.
Parameter	• id: QSPI module ID • p_cmd: control command • p_data: buffer to store the data to be read • timeout: timeout period (unit: ms)
Return Value	APP_DRV_xxx: Refer to the macro definitions in SDK_Folder\drivers\inc\app_drv_error.h.
Remarks	

3.3.2.5 app_qspi_command_receive_async

Table 3-13 app_qspi_command_receive_async API

Function Prototype	uint16_t app_qspi_command_receive_async(app_qspi_id_t id, app_qspi_command_t *p_cmd, uint8_t *p_data)
Function Description	Read data asynchronously in QSPI interrupt mode, with a control command sent first.
Parameter	• id: QSPI module ID • p_cmd: control command • p_data: buffer to store the data to be read
Return Value	APP_DRV_xxx: Refer to the macro definitions in SDK_Folder\drivers\inc\app_drv_error.h.
Remarks	

3.3.2.6 app_qspi_dma_command_receive_async

Table 3-14 app_qspi_dma_command_receive_async API

Function Prototype	uint16_t app_qspi_dma_command_receive_async(app_qspi_id_t id, app_qspi_command_t *p_cmd, uint8_t *p_data)
Function Description	Read data asynchronously in QSPI DMA mode, with a control command sent first.

Parameter	• id: QSPI module ID • p_cmd: control command • p_data: buffer to store the data to be read
Return Value	APP_DRV_xxx: Refer to the macro definitions in SDK_Folder\drivers\inc\app_drv_error.h.
Remarks	

3.3.2.7 app_qspi_command_transmit_sync

Table 3-15 app_qspi_command_transmit_sync API

Function Prototype	uint16_t app_qspi_command_transmit_sync(app_qspi_id_t id, app_qspi_command_t *p_cmd, uint8_t *p_data, uint32_t timeout)
Function Description	Transmit data synchronously in QSPI polling mode, with a control command sent first.
Parameter	• id: QSPI module ID • p_cmd: control command • p_data: buffer to store the data to be transmitted • timeout: timeout period (unit: ms)
Return Value	APP_DRV_xxx: Refer to the macro definitions in SDK_Folder\drivers\inc\app_drv_error.h.
Remarks	

3.3.2.8 app_qspi_command_transmit_async

Table 3-16 app_qspi_command_transmit_async API

Function Prototype	uint16_t app_qspi_command_transmit_async(app_qspi_id_t id, app_qspi_command_t *p_cmd, uint8_t *p_data)
Function Description	Transmit data asynchronously in QSPI interrupt mode, with a control command sent first.
Parameter	• id: QSPI module ID • p_cmd: control command • p_data: buffer to store the data to be transmitted
Return Value	APP_DRV_xxx: Refer to the macro definitions in SDK_Folder\drivers\inc\app_drv_error.h.
Remarks	

3.3.2.9 app_qspi_dma_command_transmit_async

Table 3-17 app_qspi_dma_command_transmit_async API

Function Prototype	uint16_t app_qspi_dma_command_transmit_async(app_qspi_id_t id, app_qspi_command_t *p_cmd, uint8_t *p_data)
Function Description	Transmit data asynchronously in QSPI DMA mode, with a control command sent first.

Parameter	• id: QSPI module ID • p_cmd: control command • p_data: buffer to store the data to be transmitted
Return Value	APP_DRV_xxx: Refer to the macro definitions in SDK_Folder\drivers\inc\app_drv_error.h.
Remarks	

3.3.2.10 app_qspi_transmit_sync_ex

Table 3-18 app_qspi_transmit_sync_ex API

Function Prototype	uint16_t app_qspi_transmit_sync_ex(app_qspi_id_t id, uint32_t qspi_mode, uint32_t data_width, uint8_t *p_data, uint32_t length, uint32_t timeout)
Function Description	Transmit data synchronously in QSPI polling mode; the timing mode and data width are configurable.
Parameter	• id: QSPI module ID • qspi_mode: QSPI timing mode for data transmission; options: QSPI_DATA_MODE_SPI (SPI mode), QSPI_DATA_MODE_DUALSPI (Dual SPI mode), and QSPI_DATA_MODE_QUADSPI (Quad SPI mode) • data_width: data width; options: QSPI_DATASIZE_08_BITS, QSPI_DATASIZE_16_BITS, and QSPI_DATASIZE_32_BITS • p_data: buffer to store the data to be transmitted • length: length of the data to be transmitted (unit: byte) • timeout: timeout period (unit: ms)
Return Value	APP_DRV_xxx: Refer to the macro definitions in SDK_Folder\drivers\inc\app_drv_error.h.
Remarks	

3.3.2.11 app_qspi_transmit_async_ex

Table 3-19 app_qspi_transmit_async_ex API

Function Prototype	uint16_t app_qspi_transmit_async_ex(app_qspi_id_t id, uint32_t qspi_mode, uint32_t data_width, uint8_t *p_data, uint32_t length)
Function Description	Transmit data asynchronously in QSPI interrupt mode; the timing mode and data width are configurable.
Parameter	• id: QSPI module ID • qspi_mode: QSPI timing mode for data transmission; options: QSPI_DATA_MODE_SPI (SPI mode), QSPI_DATA_MO DE_DUALSPI (Dual SPI mode), and QSPI_DATA_MODE_QUADSPI (Quad SPI mode) • data_width: data width; options: QSPI_DATASIZE_08_BITS, QSPI_DATASIZE_16_BITS, and QSPI_DATASIZE_32_BITS • p_data: buffer to store the data to be transmitted • length: length of the data to be transmitted (unit: byte)
Return Value	APP_DRV_xxx: Refer to the macro definitions in SDK_Folder\drivers\inc\app_drv_error.h.

Remarks	

3.3.2.12 app_qspi_dma_transmit_async_ex

Table 3-20 app_qspi_dma_transmit_async_ex API

Function Prototype	uint16_t app_qspi_dma_transmit_async_ex(app_qspi_id_t id, uint32_t qspi_mode, uint32_t data_width, uint8_t *p_data, uint32_t length)
Function Description	Transmit data asynchronously in QSPI DMA mode; the timing mode and data width are configurable.
Parameter	• id: QSPI module ID • qspi_mode: QSPI timing mode for data transmission; options: QSPI_DATA_MODE_SPI (SPI mode), QSPI_DATA_MO DE_DUALSPI (Dual SPI mode), and QSPI_DATA_MODE_QUADSPI (Quad SPI mode) • data_width: data width; options: QSPI_DATASIZE_08_BITS, QSPI_DATASIZE_16_BITS, and QSPI_DATASIZE_32_BITS • p_data: buffer to store the data to be transmitted • length: length of the data to be transmitted (unit: byte)
Return Value	APP_DRV_xxx: Refer to the macro definitions in SDK_Folder\drivers\inc\app_drv_error.h.
Remarks	

3.3.2.13 app_qspi_receive_sync_ex

Table 3-21 app_qspi_receive_sync_ex API

Function Prototype	uint16_t app_qspi_receive_sync_ex(app_qspi_id_t id, uint32_t qspi_mode, uint32_t data_width, uint8_t *p_data, uint32_t length , uint32_t timeout, uint32_t timeout)
Function Description	Receive data synchronously in QSPI polling mode; the timing mode and data width are configurable.
Parameter	• id: QSPI module ID • qspi_mode: QSPI timing mode for data transmission; options: QSPI_DATA_MODE_SPI (SPI mode), QSPI_DATA_MO DE_DUALSPI (Dual SPI mode), and QSPI_DATA_MODE_QUADSPI (Quad SPI mode) • data_width: data width; options: QSPI_DATASIZE_08_BITS, QSPI_DATASIZE_16_BITS, and QSPI_DATASIZE_32_BITS • p_data: buffer to store the data to be received • length: length of the data to be received (unit: byte) • timeout: timeout period (unit: ms)
Return Value	APP_DRV_xxx: Refer to the macro definitions in SDK_Folder\drivers\inc\app_drv_error.h.
Remarks	

3.3.2.14 app_qspi_receive_async_ex

Table 3-22 app_qspi_receive_async_ex API

Function Prototype	uint16_t app_qspi_receive_async_ex(app_qspi_id_t id, uint32_t qspi_mode, uint32_t data_width, uint8_t *p_data, uint32_t length)
Function Description	Receive data asynchronously in QSPI interrupt mode; the timing mode and data width are configurable.
Parameter	• id: QSPI module ID • qspi_mode: QSPI timing mode for data transmission; options: QSPI_DATA_MODE_SPI (SPI mode), QSPI_DATA_MO DE_DUALSPI (Dual SPI mode), and QSPI_DATA_MODE_QUADSPI (Quad SPI mode) • data_width: data width; options: QSPI_DATASIZE_08_BITS, QSPI_DATASIZE_16_BITS, and QSPI_DATASIZE _32_BITS • p_data: buffer to store the data to be received • length: length of the data to be received (unit: byte)
Return Value	APP_DRV_xxx: Refer to the macro definitions in SDK_Folder\drivers\inc\app_drv_error.h.
Remarks	

3.3.2.15 app_qspi_dma_receive_async_ex

Table 3-23 app_qspi_dma_receive_async_ex API

Function Prototype	uint16_t app_qspi_dma_receive_async_ex(app_qspi_id_t id, uint32_t qspi_mode, uint32_t data_width, uint8_t *p_data, uint32_t length)
Function Description	Receive data asynchronously in QSPI DMA mode; the timing mode and data width are configurable.
Parameter	• id: QSPI module ID • qspi_mode: QSPI timing mode for data transmission; options: QSPI_DATA_MODE_SPI (SPI mode), QSPI_DATA_MO DE_DUALSPI (Dual SPI mode), and QSPI_DATA_MODE_QUADSPI (Quad SPI mode) • data_width: data width; options: QSPI_DATASIZE_08_BITS, QSPI_DATASIZE_16_BITS, and QSPI_DATASIZE _32_BITS • p_data: buffer to store the data to be received • length: length of the data to be received (unit: byte)
Return Value	APP_DRV_xxx: Refer to the macro definitions in SDK_Folder\drivers\inc\app_drv_error.h.
Remarks	

3.3.3 APIs in Memory Mapped Mode

Apart from register mode, NOR Flash and PSRAM can also work in memory mapped mode by configuring the following APIs, making peripheral access simpler and more efficient.

3.3.3.1 app_qspi_config_memory_mappped

Table 3-24 app_qspi_config_memory_mappped API

Function Prototype	bool app_qspi_config_memory_mappped(app_qspi_id_t id, app_qspi_mmap_device_t dev);

Function Description	Set the device to memory mapped mode for transmitting data blocks.
Parameter	• id: QSPI module ID • dev: Specify the device type to work in memory mapped mode.
Return Value	true/false
Remarks	

app_qspi_mmap_device_t definition and members are detailed as follows.

```
typedef struct {
    app_qspi_device_e                       dev_type;
    app_qspi_psram_mmap_wr_cmd_e            psram_wr;
    union {
        app_qspi_flash_mmap_rd_cmd_e        flash_rd;
        app_qspi_psram_mmap_rd_cmd_e        psram_rd;
    } rd;
    void * set;
} app_qspi_mmap_device_t;
```

dev_type

- APP_QSPI_DEVICE_FLASH: NOR Flash
- APP_QSPI_DEVICE_PSRAM: PSRAM

psram_wr

Specify the write command for PSRAM when “dev_type” is configured as “APP_QSPI_DEVICE_PSRAM”. Options include

- PSRAM_MMAP_CMD_QWRITE_02H
- PSRAM_MMAP_CMD_QWRITE_38H

rd

Specify the read command for PSRAM/NOR Flash. When NOR Flash is adopted, options for the read command flash_rd include

- FLASH_MMAP_CMD_DREAD_3BH: 3B command in Dual SPI mode
- FLASH_MMAP_CMD_2READ_BBH: BB command in Dual SPI mode
- FLASH_MMAP_CMD_QREAD_6BH: 6B command in Quad SPI mode
- FLASH_MMAP_CMD_4READ_EBH: EB command in Quad SPI mode

When PSRAM is adopted, options for the read command psram_rd include

- PSRAM_MMAP_CMD_QREAD_0BH: 0B command in Quad SPI mode
- PSRAM_MMAP_CMD_QREAD_EBH: EB command in Quad SPI mode

3.3.3.2 app_qspi_active_memory_mappped

Table 3-25 app_qspi_active_memory_mappped API

Function Prototype	bool app_qspi_active_memory_mappped(app_qspi_id_t id, bool is_active)
Function Description	Enable/Disable memory mapped mode.
Parameter	• id: QSPI module ID • is_active: Enable/Disable memory mapped mode.
Return Value	true/false
Remarks	

3.3.3.3 app_qspi_get_xip_base_address

Table 3-26 app_qspi_get_xip_base_address API

Function Prototype	uint32_t app_qspi_get_xip_base_address(app_qspi_id_t id)
Function Description	Obtain the base address in system bus mapped with QSPI memory.
Parameter	id: QSPI module ID
Return Value	Base address in system bus mapped with QSPI memory
Remarks	

3.3.3.4 app_qspi_mmap_set_endian_mode

Table 3-27 app_qspi_mmap_set_endian_mode API

Function Prototype	bool app_qspi_mmap_set_endian_mode(app_qspi_id_t id, app_qspi_mmap_endian_mode_e mode)
Function Description	Configure the data endian mode for read operations in memory mapped mode.
Parameter	• id: QSPI module ID • mode: data endian mode for read operations
Return Value	true/false
Remarks	

3.3.4 Special APIs

Special APIs are the ones combining key features of QSPI and DMA modules, making them an efficient choice for such scenarios as display refresh.

3.3.4.1 app_qspi_async_draw_screen

Table 3-28 app_qspi_async_draw_screen API

Function Prototype	bool app_qspi_async_draw_screen (app_qspi_id_t screen_id, app_qspi_id_t storage_id, const app_qspi_screen_command_t * const p_screen_cmd,

	const app_qspi_screen_info_t * const p_screen_info, app_qspi_screen_scroll_t * p_scroll_config, bool is_first_call)
Function Description	Perform asynchronous display refresh.
Parameter	• screen_id: display ID • storage_id: ID of the QSPI memory that stores materials • p_screen_cmd: pointer to the display refresh control command • p_screen_info: display information • p_scroll_config: display scrolling control information • is_first_call: To call this API, set this parameter to “true”.
Return Value	true/false
Remarks	• storage_id shall be set to “APP_STORAGE_RAM_ID” when graphic materials are not stored in the QSPI memory. • storage_id and screen_id cannot be the same. • scrn_pixel_depth of the parameter “p_screen_info” can be set to “2” only. That is, only 16-bit display output is supported by this API. • The display drive API is extracted in SDK_Folder\drivers\inc\app_graphics_qspi.h.

3.3.4.2 app_qspi_async_veri_draw_screen

Table 3-29 app_qspi_async_veri_draw_screen API

Function Prototype	bool app_qspi_async_veri_draw_screen(app_qspi_id_t screen_id, app_qspi_id_t storage_id, const app_qspi_screen_command_t * const p_screen_cmd, const app_qspi_screen_info_t * const p_screen_info, app_qspi_screen_veri_link_scroll_t * p_link_scroll, bool is_first_call)
Function Description	Perform asynchronous (vertical) chaining display refresh.
Parameter	• screen_id: display ID • storage_id: ID of the QSPI memory that stores materials • p_screen_cmd: pointer to the display refresh control command • p_screen_info: display information • p_link_scroll: pointer to the chaining scrolling parameters • is_first_call: To call this API, set this parameter to “true”.
Return Value	• true: Succeeded • false: Failed
Remarks	• storage_id shall be set to “APP_STORAGE_RAM_ID” when graphic materials are not stored in the QSPI memory.

	• storage_id and screen_id cannot be the same. • scrn_pixel_depth of the parameter “p_screen_info” can be set to “2” only. That is, only 16-bit display output is supported by this API. • The display drive API is extracted in SDK_Folder\drivers\inc\app_graphics_qspi.c.

3.3.4.3 app_qspi_mmap_blit_image

Table 3-30 app_qspi_mmap_blit_image API

Function Prototype	bool app_qspi_mmap_blit_image(app_qspi_id_t storage_id, blit_image_config_t * p_blit_config, blit_xfer_type_e xfer_type)
Function Description	Transmit two-dimensional data through DMA.
Parameter	• storage_id: ID of the QSPI memory that stores materials • p_blit_config: blit configuration • xfer_type: transmission type; options: SG and LLP
Return Value	• true: Succeeded • false: Failed
Remarks	• The display drive API is extracted in SDK_Folder\drivers\inc\app_graphics_qspi.c. • The macro QSPI_BLIT_RECT_IMAGE_SUPPORT in *app_qspi_user_config.h* shall be set to a value larger than 0 (default: 0) before this API is called.

3.3.4.4 app_qspi_async_llp_draw_block

Table 3-31 app_qspi_async_llp_draw_block API

Function Prototype	bool app_qspi_async_llp_draw_block(app_qspi_id_t screen_id, app_qspi_id_t storage_id, const app_qspi_screen_command_t *const p_screen_cmd, const app_qspi_screen_info_t *const p_screen_info, app_qspi_screen_block_t *p_block_info, bool is_first_call)
Function Description	Refresh the display block areas by means of DMA LLP (DMA chain). Each node of the linked list corresponds to a line of data in the graphic data block(s) to be refreshed.
Parameter	• screen_id: display ID • storage_id: ID of the QSPI memory that stores materials • p_screen_cmd: pointer to the display refresh control command • p_screen_info: display information • p_block_info: graphic data block(s) to be refreshed

	• is_first_call: When this API is called for the first time, set this parameter to “true”, which indicates beginning of a frame of graphic materials and a command shall be transmitted.
Return Value	• true: Succeeded • false: Failed
Remarks	When data transmission completes, the APP_QSPI_EVT_TX_CPLT event is triggered.

3.4 QSPI Operational Recommendations

1. System peripheral clock that is the QSPI clock source serves as the baseline of QSPI clock prescaler value. The QSPI clock prescaler value shall be an even number in the range of 2–65535.
2. For data transmission of SRAM or QSPI devices in memory mapped mode, when the data to be transmitted is less than 1 KB, the memcpy function is recommended in most cases; DMA transmission is recommended when the data size is larger than 1 KB. DMA has obvious advantages on transmitting a large amount of data.
3. Before the memcpy/memset function is adopted to access data in memory mapped mode, you are recommended to compile the system standard library in Keil instead of MicroLib for the sake of more efficient access, because the former can better activate the bus burst transfer.
4. Some QSPI sequences require enough Tcsu (CS Setup Delay Time) to ensure stable data access. You can
   - Set the clock mode to Clock Mode 3 if supported. Generally, Clock Mode 3 has longer Tcsu than Clock Mode 0.
   - Increase Tcsu by configuring the specific register.
5. It is necessary for QSPI PSRAM to release CS regularly to perform self-refresh of the internal data retention circuit. The longest CS time is defined as “t_{CEM}”, which shall be complied with strictly, especially for write operations.
   - For write operations in memory mapped mode, t_{CEM} is taken into account in design, and no configuration is required.
   - For write operations in register mode, t_{CEM} is managed by the driver layer. QSPI PSRAM is not recommended in this mode due to the complex API design.
6. QSPI supports data pre-fetch mode, in which optimal read efficiency of QSPI can be ensured in read operations. Note that this mode can only work with DMA, and cannot be enabled during CPU access.
7. In GR5525 SoC design, differentiated optimization to each QSPI/DMA FIFO is performed, making it feasible to connect peripherals randomly in most cases. To highlight the optimal efficiency of GR5525 series, recommended connection schemes are as follows:
   - QSPI0: Connect Flash preferentially; work with channel 0 of DAM0.
   - QSPI1: Connect PSRAM preferentially; work with channel 0 of DAM0/DAM1.
   - QSPI2: Connect a display device preferentially; work with channel 0 of DAM1.
   - It is recommended to allocate channel 0 of DAM0/DAM1 to peripherals with high throughput such as QSPI0/QSPI1/QSPI2, and not to adopt the two channels simultaneously.

4 Display Control and Rendering Logics

4.1 Display Timing

GR5525 provides QSPI, Display SPI, and SPIM modules for external displays, primarily supporting display interface specifications such as MIPI Display Bus Interface (DBI) Type-C, which covers the following timing protocols:

- SPI (8-bit version)

  Include three signal lines: SPI_CS (chip enable), SPI_SCLK (serial clock), and SPI_SD (serial data output). Control commands, addresses, and color data are all sent through SPI_SD.

- 4-line SPI (extended DCX signal line)

  Include four signal lines: SPI_CS (chip enable), SPI_SCLK (serial clock), SPI_SD (serial data output), and SPI_DC (data/command indicator). The command word consists of 8 bits. The data word consists of several bits of pixel data.

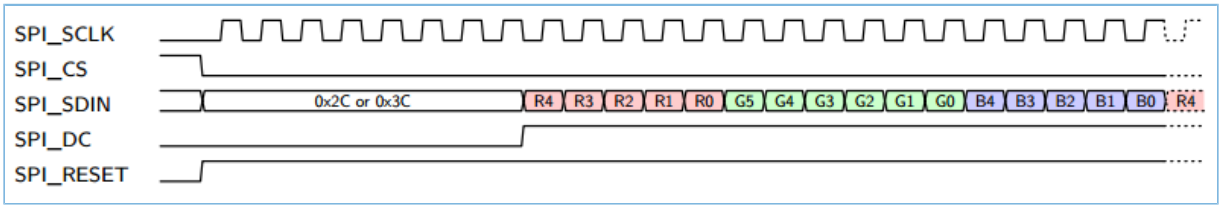


Figure 4-1 Reference timing of 4-line SPI – RGB565 – Option0 format

- Dual SPI

  Include four signal lines: SPI_CS (chip enable), SPI_SCLK (serial clock), as well as SPI_SD and SPI_SD1 (serial data output). The command word is always transmitted through SPI_SD, and the data is transmitted through SPI_SD and SPI_SD1. The command word consists of 9 bits: 1-bit data/command indicator and 8-bit command. The data word consists of 1-bit data/command indicator and several bits of pixel data.

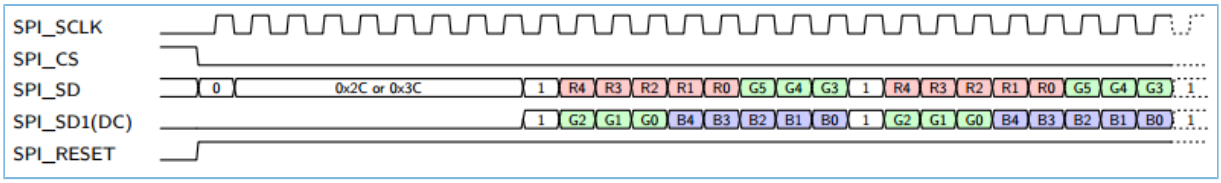


Figure 4-2 Reference timing of Dual SPI – RGB565 – Option0 format

- Quad SPI

  Include six signal lines: SPI_CS (chip enable), SPI_SCLK (serial clock), as well as SPI_SD, SPI_SD1, SPI_SD2, and SPI_SD3 (serial data output). The command type is always transmitted through SPI_SD or SPI_SDx. QSPI timing generally consists of 8-bit command header, 24-bit command, and several bits of pixel data.

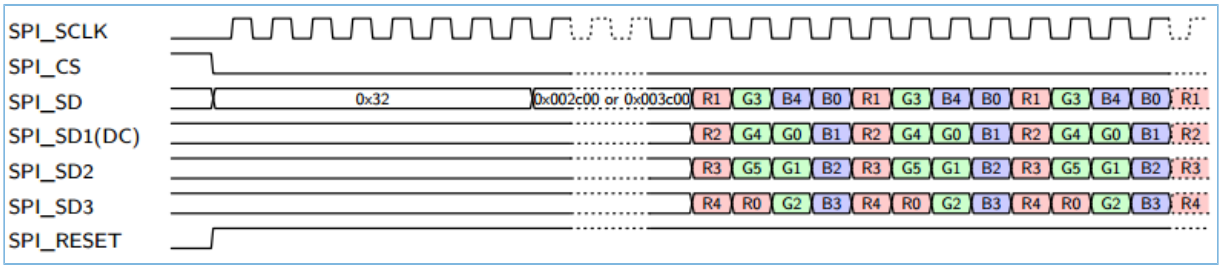


Figure 4-3 Reference timing of Quad SPI – RGB565 – Option0 format

4.2 Color Format

4.2.1 Color and Color Depth

Color (RGB color) is a triplet composed of three components: red, green, and blue. The components determine color intensity or brightness. Different color formats represent different color depths of components in an RGB tuple. Color depth refers to the number of bits that are stored in the frame buffer and used to describe each color. It can also be expressed as bits per pixel (bpp). Higher color depth means finer color details and more color choices.

GR5525 supports RGB888 and RGB565, corresponding to the following color components and color depths:

1. RGB888 (24-bit color depth): Each color component is represented as 8 bits (24 bits/24 bpp in total). Therefore, RGB888 can represent 16777216 (2^{24}) colors. The component ranges from 0 to 255, with a larger value indicating greater color intensity/brightness. That is, "0" represents no color, and 255 indicates the color is at the maximum value.
2. RGB565 (16-bit color depth): The R and B components are each represented as 5 bits, with the component range of 0 to 31; the G component is represented as 6 bits, with the component range of 0 to 63. Therefore, RGB565 can represent 65,536 (2^{16}) colors.

As shown below, the RGB color is (0, 0, 0) for pure black, (255, 255, 255) for pure white, and (0, 255, 0) for bright green.

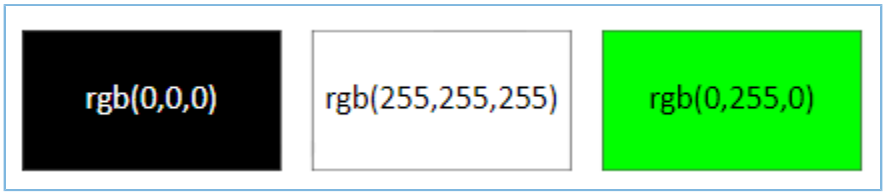
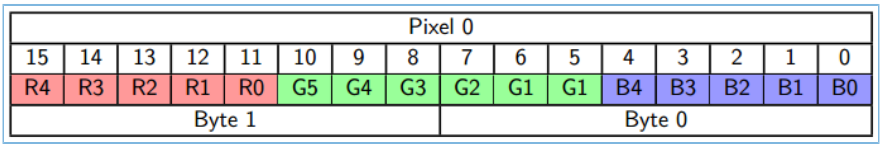


Figure 4-4 Diagram of RGB tuple components

The 1 bpp format can only represent 2 (2^1) colors. Therefore, it is applicable to applications with black and white only.

For GR5525 in wearable applications, RGB565 (65K colors) is recommended.



Pixel 0															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R4	R3	R2	R1	R0	G5	G4	G3	G2	G1	G1	B4	B3	B2	B1	B0
Byte 1								Byte 0							

Figure 4-5 RGB565 16-bit format

4.2.2 Alpha Blending

Alpha blending is a frequently used image processing technique to achieve special image effects and image blending. It can overlay multiple images together by controlling the transparency of each pixel in the image. The transparency value of each pixel is used to determine the ratio of blending between two images, to achieve different levels of transparency.

With alpha blending, various visual effects such as semi-transparent masks, image gradients, and shadows can be created. By adjusting the transparency of different images, you can achieve diverse effects.

Alpha blending of images requires a significant amount of computational cycles and can occupy CPU resources for a long time in the rendering process. Therefore, it is recommended that UI/UX engineers use design techniques to achieve good graphic effects or use alpha blending calculations only for special scenarios or certain images.

4.3 Frame Buffer

4.3.1 Basic Concepts

The frame buffer is a memory section used to store rendering structures, typically allocated from SRAM. The frame buffer serves as the pivotal hub for interaction between rendering logic and display logic, with the interaction process as follows:

1. Computing devices (such as CPU) load image resources from peripherals (such as Flash) into the frame buffer; then, perform calculations through the GUI framework and following the designed UI/UX principles.
2. Once frame image calculations are completed, modules (such as DMA) transfer the data from the frame buffer through display peripheral interfaces (such as QSPI) to an external screen for display.

The basic unit of the frame buffer is pixel. The frame buffer can generally be organized in a two-dimensional array data structure, constructed with the following three main parameters:

- Width: the number of horizontal pixels in the display area. Generally, this value is the same as the width of the display device in the product, except for special scenarios. For example, the width of the display device might be odd; however, the frame buffer width might be set to an even number to make computation easier.
- Height: the number of vertical pixels in the display area. This value depends on the GUI framework and SRAM and generally does not exceed the height of the display device.
- Depth: the pixel depth of the color format, indicating how many bytes a pixel occupies. This value can serve as the basic unit for the two-dimensional array elements. For example, the depth of RGB565 is 2 bytes.

Space occupation of the frame buffer is determined by the width, height, and depth. The formula is as follows: Frame buffer size = Width x Height x Depth.

The frame buffer can also be indexed using plane coordinates, with the upper-left corner as the coordinate origin, as shown below:

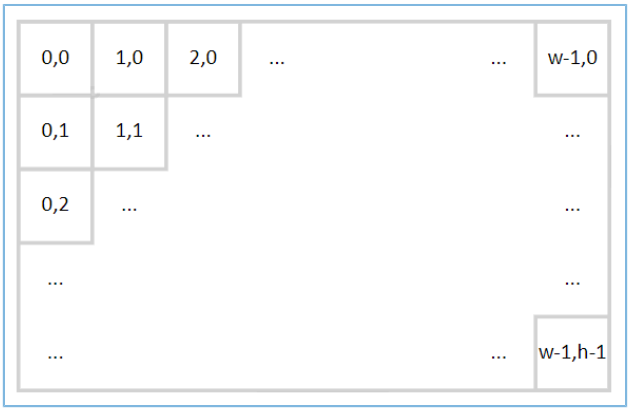


Figure 4-6 Coordinate index example of frame buffer

4.3.2 Frame Buffer Quantity

The number of frame buffers required in the graphic software architecture is determined by parameters such as display resolution, color format and depth, available SRAM space, and display frame rate. Generally, reference designs are as follows:

- One complete buffer

  This logical architecture design is the simplest: A frame buffer needs to buffer all pixels to be transmitted to the display; rendering and display refresh are performed in sequence, meaning that the display refresh logic needs to wait during rendering, and vice versa.

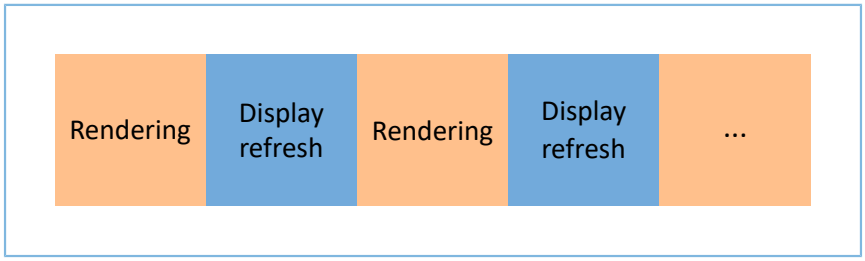


Figure 4-7 Interaction logic between display refresh and rendering (with one complete buffer)

  This design can be considered if there is no strict requirement of display frame rate or if there is sufficient SRAM to provide a complete frame buffer due to small screen. Display refresh and rendering cannot be performed simultaneously. Therefore, this architecture cannot achieve better frame rate performance and thus is less commonly used.

- Two complete buffers

  Display refresh and rendering can be performed simultaneously, and chip performance can be fully utilized for parallel tasks, effectively improving the refresh frame rate. This approach requires a large SRAM size. Given the limited SRAM resources of GR5525, it is impossible to provide two complete frame buffers for displays above a certain resolution.

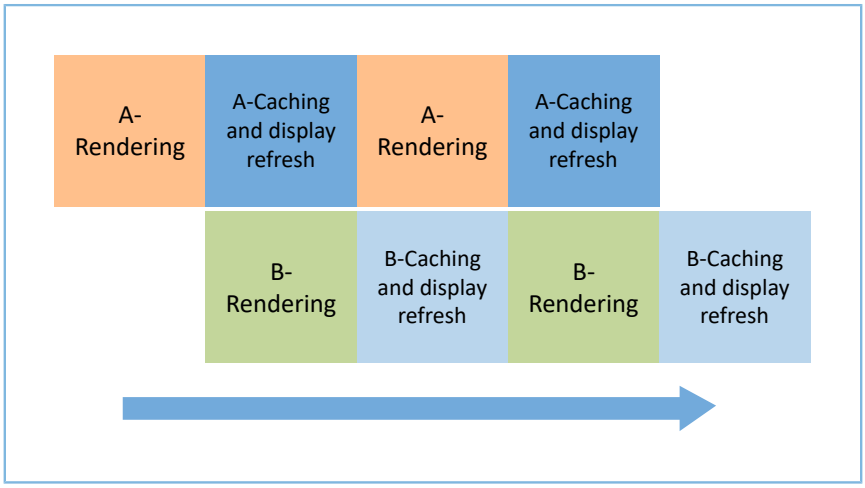


Figure 4-8 Interaction logic between display refresh and rendering (with two complete buffers)

- Two partial buffers

  In this buffer architecture, rendering and display refresh can be performed simultaneously, to improve frame rate performance and save SRAM resources.

  Each rendering and refresh cover an area of (Display width x Partial height x Pixel depth). Several renderings and refreshes are needed to display a screen of data, making the software architecture more complex.

  This design has certain limitations if the display itself does not come with SRAM.

4.3.3 Reference Design

4.3.3.1 GUI Framework

When GR5525 is applied to wearable products and other display devices, users can build a GUI framework based on product requirements. The following shall be considered:

- Basic periodic scheduling framework
- Parallel design for rendering and display refresh
- Data processing for event input devices (Touch/Key)
- Whether to build simple graphic components
- Reasonable utilization of resources
- Maximizing chip performance and utilization rate

Users can also adopt third-party open-source GUI frameworks, such as LVGL. LVGL is a graphic framework designed for embedded devices, capable of achieving good display effects on chips with limited resources. However, the native implementation of UI controls in LVGL still consumes a significant amount of computational power and SRAM resources of embedded devices, and optimization is required according to user needs. GR5525 SDK provides deeply optimized LVGL, which can achieve better frame rates for wearable devices, available for users to refer to in software design.

4.3.3.2 Watch Demo Project

GR5525 SDK provides an example project (available in `SDK_Folder\projects\peripheral\graphics\graphics_lvgl_831_std_demo`) for wearable devices. You can download the project to GR5525 Starter Kit Board through GProgrammer. The example project mainly covers the following aspects for user reference:

Note:

SDK_Folder is the root directory of GR5525 SDK.

- Porting and optimization of LVGL
- UI/UX interface reference design based on LVGL
- Extended custom design and window management system
- Peripheral drivers and usage reference
- Usage of Bluetooth functionalities