



GR551x HAL and LL Drivers User Manual

Version: 1.7

Release Date: 2020-12-18

Copyright © 2020 Shenzhen Goodix Technology Co., Ltd. All rights reserved.

Any excerption, backup, modification, translation, transmission or commercial use of this document or any portion of this document, in any form or by any means, without the prior written consent of Shenzhen Goodix Technology Co., Ltd is prohibited.

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: 2F. & 13F., Tower B, Tengfei Industrial Building, Futian Free Trade Zone, Shenzhen, China

TEL: +86-755-33338828

FAX: +86-755-33338099

Website: www.goodix.com

Preface

Purpose

This document introduces the peripheral driver architecture, driver files, Application Programming Interfaces (APIs), and driver naming rules of GR551x SoCs. The peripheral drivers comprise Hardware Abstraction Layer (HAL) drivers and Low Layer (LL) drivers. The document elaborates on the usage, data structures, and APIs of HAL and LL drivers, aiming to help developers quickly use HAL APIs to enable interactions between upper-layer applications and low-layer peripherals. The document also enables developers to utilize LL APIs in driver porting and adaptation in a real-time operating system (RTOS).

Audience

This document is intended for:

- GR551x user
- GR551x developer
- GR551x tester
- Technical writer

Release Notes

This document is the fifth release of *GR551x HAL and LL Drivers User Manual*, corresponding to GR551x SoC series.

Revision History

| Version | Date | Description |
|---------|------------|--|
| 1.0 | 2019-12-08 | Initial release |
| 1.3 | 2020-03-16 | Deleted CGC driver; modified calendar APIs and structures. |
| 1.5 | 2020-05-30 | <ul style="list-style-type: none"> • Added peripheral configuration registers in sleep mode and during wakeup; added description on disabling a specified AON GPIO API to wake up the system. • Deleted the memory power control APIs and the AON GPIO wakeup interrupt handler APIs in “Section 2.19.3 PWR Driver APIs”. |
| 1.6 | 2020-06-30 | <ul style="list-style-type: none"> • Modified the ADC reference voltages in “Section 2.10 HAL ADC Generic Driver” and “Section 3.4.1 ADC Driver Structure”. • Updated the usage of polling/interrupt/DMA mode, as well as some HMAC driver APIs and parameters in “Section 2.14 HAL HMAC Generic Driver”. • Updated some parameters in “Section 2.22.3 Calendar Driver Structures”. |
| 1.7 | 2020-12-18 | <ul style="list-style-type: none"> • Added descriptions about PWM alignment in “Section 2.18.3.2 pwm_init_t” and “Section 3.11.1.2 ll_pwm_init_t”. • Added limitations on using RNG_OUTPUT_FR0_S0 in “Section 2.25.3.1 rng_init_t” and “Section 3.16.1.1 ll_rng_init_t”. |

Contents

| | |
|--|-----------|
| Preface..... | I |
| 1 Overview..... | 1 |
| 1.1 GR551x Peripheral Drivers..... | 1 |
| 1.1.1 HAL Drivers..... | 1 |
| 1.1.2 LL Drivers..... | 2 |
| 1.2 File Classification..... | 2 |
| 1.2.1 Driver Files..... | 2 |
| 1.2.1.1 Header Files..... | 2 |
| 1.2.1.2 HAL Driver Files..... | 2 |
| 1.2.1.3 LL Driver Files..... | 4 |
| 1.2.2 User-Application Files..... | 4 |
| 1.3 API Classification..... | 5 |
| 1.3.1 Generic APIs..... | 5 |
| 1.3.1.1 HAL Generic APIs..... | 5 |
| 1.3.1.2 LL Generic APIs..... | 5 |
| 1.3.2 Extension APIs..... | 6 |
| 1.4 Driver Naming Rules..... | 6 |
| 1.4.1 General Naming Rules..... | 6 |
| 1.4.2 Naming Rules of HAL Driver APIs..... | 7 |
| 1.4.3 Naming Rules of LL Driver APIs..... | 9 |
| 1.5 Data Structure..... | 11 |
| 1.5.1 Peripheral Handle Structure..... | 11 |
| 1.5.2 Initialization Structure..... | 12 |
| 1.5.3 Configuration Structure..... | 12 |
| 2 HAL Drivers..... | 14 |
| 2.1 Introduction..... | 14 |
| 2.1.1 HAL Common Resources..... | 14 |
| 2.1.2 How to Use HAL Drivers..... | 14 |
| 2.1.2.1 HAL Driver Initialization..... | 16 |
| 2.1.2.2 HAL Driver I/O Operations..... | 17 |
| 2.1.2.3 Timeout Detection and Error Check..... | 20 |
| 2.2 HAL Cortex Generic Driver..... | 23 |
| 2.2.1 Cortex Driver Functionalities..... | 23 |
| 2.2.2 How to Use Cortex Driver..... | 23 |
| 2.2.3 Cortex Driver APIs..... | 23 |
| 2.2.3.1 hal_nvic_set_priority_grouping..... | 24 |
| 2.2.3.2 hal_nvic_set_priority..... | 24 |
| 2.2.3.3 hal_nvic_enable_irq..... | 25 |

| | |
|---|----|
| 2.2.3.4 hal_nvic_disable_irq..... | 25 |
| 2.2.3.5 hal_nvic_system_reset..... | 25 |
| 2.2.3.6 hal_systick_config..... | 25 |
| 2.2.3.7 hal_nvic_get_priority_grouping..... | 26 |
| 2.2.3.8 hal_nvic_get_priority..... | 26 |
| 2.2.3.9 hal_nvic_set_pending_irq..... | 27 |
| 2.2.3.10 hal_nvic_get_pending_irq..... | 27 |
| 2.2.3.11 hal_nvic_clear_pending_irq..... | 28 |
| 2.2.3.12 hal_nvic_get_active..... | 28 |
| 2.2.3.13 hal_systick_clk_source_config..... | 28 |
| 2.2.3.14 hal_systick_irq_handler..... | 28 |
| 2.2.3.15 hal_systick_callback..... | 29 |
| 2.3 HAL System Driver..... | 29 |
| 2.3.1 System Driver Functionalities..... | 29 |
| 2.3.2 How to Use System Driver..... | 29 |
| 2.3.3 System Driver APIs..... | 29 |
| 2.3.3.1 hal_init..... | 30 |
| 2.3.3.2 hal_deinit..... | 30 |
| 2.3.3.3 hal_msp_init..... | 31 |
| 2.3.3.4 hal_msp_deinit..... | 31 |
| 2.3.3.5 hal_init_tick..... | 31 |
| 2.3.3.6 hal_suspend_tick..... | 31 |
| 2.3.3.7 hal_resume_tick..... | 32 |
| 2.3.3.8 hal_get_hal_version..... | 32 |
| 2.4 HAL GPIO Generic Driver..... | 32 |
| 2.4.1 GPIO Driver Functionalities..... | 32 |
| 2.4.2 How to Use GPIO Driver..... | 32 |
| 2.4.3 GPIO Driver Structures..... | 33 |
| 2.4.3.1 gpio_init_t..... | 33 |
| 2.4.4 GPIO Driver APIs..... | 34 |
| 2.4.4.1 hal_gpio_init..... | 34 |
| 2.4.4.2 hal_gpio_deinit..... | 35 |
| 2.4.4.3 hal_gpio_read_pin..... | 36 |
| 2.4.4.4 hal_gpio_write_pin..... | 36 |
| 2.4.4.5 hal_gpio_toggle_pin..... | 37 |
| 2.4.4.6 hal_gpio_exti_irq_handler..... | 38 |
| 2.4.4.7 hal_gpio_exti_callback..... | 38 |
| 2.5 HAL GPIO Extension Driver..... | 39 |
| 2.5.1 GPIO Driver Defines..... | 39 |
| 2.5.1.1 GPIO Multiplexing Selection..... | 39 |
| 2.6 HAL AON GPIO Generic Driver..... | 47 |
| 2.6.1 AON GPIO Driver Functionalities..... | 47 |

| | |
|--|----|
| 2.6.2 How to Use AON GPIO Driver..... | 47 |
| 2.6.3 AON GPIO Driver Structures..... | 47 |
| 2.6.3.1 aon_gpio_init_t..... | 47 |
| 2.6.4 AON GPIO Driver APIs..... | 48 |
| 2.6.4.1 hal_aon_gpio_init..... | 49 |
| 2.6.4.2 hal_aon_gpio_deinit..... | 49 |
| 2.6.4.3 hal_aon_gpio_read_pin..... | 49 |
| 2.6.4.4 hal_aon_gpio_write_pin..... | 50 |
| 2.6.4.5 hal_aon_gpio_toggle_pin..... | 51 |
| 2.6.4.6 hal_aon_gpio_irq_handler..... | 51 |
| 2.6.4.7 hal_aon_gpio_callback..... | 51 |
| 2.7 HAL AON GPIO Extension Driver..... | 52 |
| 2.7.1 AON GPIO Driver Defines..... | 52 |
| 2.7.1.1 AON GPIO Multiplexing Selection..... | 52 |
| 2.8 HAL MSIO Generic Driver..... | 53 |
| 2.8.1 MSIO Driver Functionalities..... | 54 |
| 2.8.2 How to Use MSIO Driver..... | 54 |
| 2.8.3 MSIO Driver Structures..... | 54 |
| 2.8.3.1 msio_init_t..... | 54 |
| 2.8.4 MSIO Driver APIs..... | 55 |
| 2.8.4.1 hal_msio_init..... | 55 |
| 2.8.4.2 hal_msio_deinit..... | 55 |
| 2.8.4.3 hal_msio_read_pin..... | 56 |
| 2.8.4.4 hal_msio_write_pin..... | 56 |
| 2.8.4.5 hal_msio_toggle_pin..... | 57 |
| 2.9 HAL MSIO Extension Driver..... | 57 |
| 2.9.1 MSIO Driver Defines..... | 57 |
| 2.9.1.1 MSIO Multiplexing Selection..... | 57 |
| 2.10 HAL ADC Generic Driver..... | 59 |
| 2.10.1 ADC Driver Functionalities..... | 59 |
| 2.10.2 How to Use ADC Driver..... | 59 |
| 2.10.3 ADC Driver Structures..... | 60 |
| 2.10.3.1 adc_init_t..... | 60 |
| 2.10.3.2 adc_handle_t..... | 61 |
| 2.10.4 ADC Driver APIs..... | 62 |
| 2.10.4.1 hal_adc_init..... | 63 |
| 2.10.4.2 hal_adc_deinit..... | 63 |
| 2.10.4.3 hal_adc_msp_init..... | 63 |
| 2.10.4.4 hal_adc_msp_deinit..... | 64 |
| 2.10.4.5 hal_adc_poll_for_conversion..... | 64 |
| 2.10.4.6 hal_adc_start_dma..... | 64 |
| 2.10.4.7 hal_adc_stop_dma..... | 65 |

| | |
|---|----|
| 2.10.4.8 hal_adc_conv_cplt_callback..... | 65 |
| 2.10.4.9 hal_adc_get_state..... | 66 |
| 2.10.4.10 hal_adc_get_error..... | 66 |
| 2.10.4.11 hal_adc_set_dma_threshold..... | 66 |
| 2.10.4.12 hal_adc_get_dma_threshold..... | 67 |
| 2.10.4.13 hal_adc_suspend_reg..... | 67 |
| 2.10.4.14 hal_adc_resume_reg..... | 67 |
| 2.11 HAL DMA Generic Driver..... | 68 |
| 2.11.1 DMA Driver Functionalities..... | 68 |
| 2.11.2 How to Use DMA Driver..... | 68 |
| 2.11.3 DMA Driver Structures..... | 69 |
| 2.11.3.1 dma_init_t..... | 69 |
| 2.11.3.2 dma_handle_t..... | 71 |
| 2.11.4 DMA Driver APIs..... | 73 |
| 2.11.4.1 hal_dma_init..... | 74 |
| 2.11.4.2 hal_dma_deinit..... | 74 |
| 2.11.4.3 hal_dma_start..... | 74 |
| 2.11.4.4 hal_dma_start_it..... | 75 |
| 2.11.4.5 hal_dma_abort..... | 75 |
| 2.11.4.6 hal_dma_abort_it..... | 75 |
| 2.11.4.7 hal_dma_poll_for_transfer..... | 76 |
| 2.11.4.8 hal_dma_irq_handler..... | 76 |
| 2.11.4.9 hal_dma_register_callback..... | 76 |
| 2.11.4.10 hal_dma_unregister_callback..... | 77 |
| 2.11.4.11 hal_dma_get_state..... | 77 |
| 2.11.4.12 hal_dma_get_error..... | 77 |
| 2.11.4.13 hal_dma_suspend_reg..... | 78 |
| 2.11.4.14 hal_dma_resume_reg..... | 78 |
| 2.12 HAL DUAL TIMER Generic Driver..... | 78 |
| 2.12.1 DUAL TIMER Driver Functionalities..... | 78 |
| 2.12.2 How to Use DUAL TIMER Driver..... | 79 |
| 2.12.3 DUAL TIMER Driver Structures..... | 79 |
| 2.12.3.1 dual_timer_init_t..... | 79 |
| 2.12.3.2 dual_timer_handle_t..... | 80 |
| 2.12.4 DUAL TIMER Driver APIs..... | 80 |
| 2.12.4.1 hal_dual_timer_base_init..... | 81 |
| 2.12.4.2 hal_dual_timer_base_deinit..... | 81 |
| 2.12.4.3 hal_dual_timer_base_msp_init..... | 82 |
| 2.12.4.4 hal_dual_timer_base_msp_deinit..... | 82 |
| 2.12.4.5 hal_dual_timer_base_start..... | 82 |
| 2.12.4.6 hal_dual_timer_base_stop..... | 83 |
| 2.12.4.7 hal_dual_timer_base_start_it..... | 83 |

| | |
|---|-----|
| 2.12.4.8 hal_dual_timer_base_stop_it..... | 83 |
| 2.12.4.9 hal_dual_timer_set_config..... | 83 |
| 2.12.4.10 hal_dual_timer_irq_handler..... | 84 |
| 2.12.4.11 hal_dual_timer_period_elapsed_callback..... | 84 |
| 2.12.4.12 hal_dual_timer_get_state..... | 84 |
| 2.13 HAL AES Generic Driver..... | 85 |
| 2.13.1 AES Driver Functionalities..... | 85 |
| 2.13.2 How to Use AES Driver..... | 85 |
| 2.13.2.1 Initialization..... | 85 |
| 2.13.2.2 Encryption and Decryption in ECB Mode..... | 86 |
| 2.13.2.3 Encryption and Decryption in CBC Mode..... | 86 |
| 2.13.3 AES Driver Structures..... | 87 |
| 2.13.3.1 aes_init_t..... | 88 |
| 2.13.3.2 aes_handle_t..... | 88 |
| 2.13.4 AES Driver APIs..... | 90 |
| 2.13.4.1 hal_aes_init..... | 91 |
| 2.13.4.2 hal_aes_deinit..... | 91 |
| 2.13.4.3 hal_aes_msp_init..... | 91 |
| 2.13.4.4 hal_aes_msp_deinit..... | 92 |
| 2.13.4.5 hal_aes_ecb_encrypt..... | 92 |
| 2.13.4.6 hal_aes_ecb_decrypt..... | 92 |
| 2.13.4.7 hal_aes_cbc_encrypt..... | 93 |
| 2.13.4.8 hal_aes_cbc_decrypt..... | 93 |
| 2.13.4.9 hal_aes_ecb_encrypt_it..... | 94 |
| 2.13.4.10 hal_aes_ecb_decrypt_it..... | 94 |
| 2.13.4.11 hal_aes_cbc_encrypt_it..... | 95 |
| 2.13.4.12 hal_aes_cbc_decrypt_it..... | 95 |
| 2.13.4.13 hal_aes_ecb_encrypt_dma..... | 95 |
| 2.13.4.14 hal_aes_ecb_decrypt_dma..... | 96 |
| 2.13.4.15 hal_aes_cbc_encrypt_dma..... | 96 |
| 2.13.4.16 hal_aes_cbc_decrypt_dma..... | 97 |
| 2.13.4.17 hal_aes_abort..... | 97 |
| 2.13.4.18 hal_aes_abort_it..... | 97 |
| 2.13.4.19 hal_aes_irq_handler..... | 98 |
| 2.13.4.20 hal_aes_done_callback..... | 98 |
| 2.13.4.21 hal_aes_error_callback..... | 98 |
| 2.13.4.22 hal_aes_abort_cplt_callback..... | 99 |
| 2.13.4.23 hal_aes_get_state..... | 99 |
| 2.13.4.24 hal_aes_get_error..... | 99 |
| 2.13.4.25 hal_aes_set_timeout..... | 100 |
| 2.13.4.26 hal_aes_suspend_reg..... | 100 |
| 2.13.4.27 hal_aes_resume_reg..... | 100 |

| | |
|--|-----|
| 2.14 HAL HMAC Generic Driver..... | 100 |
| 2.14.1 HMAC Driver Functionalities..... | 100 |
| 2.14.2 How to Use HMAC Driver..... | 101 |
| 2.14.2.1 Initialization..... | 101 |
| 2.14.2.2 Calculate Message Digests by Using SHA-256..... | 101 |
| 2.14.2.3 Calculate Message Signatures by Using HMAC..... | 102 |
| 2.14.3 HMAC Driver Structures..... | 103 |
| 2.14.3.1 hmac_init_t..... | 103 |
| 2.14.3.2 hmac_handle_t..... | 103 |
| 2.14.4 HMAC Driver APIs..... | 105 |
| 2.14.4.1 hal_hmac_init..... | 105 |
| 2.14.4.2 hal_hmac_deinit..... | 106 |
| 2.14.4.3 hal_hmac_msp_init..... | 106 |
| 2.14.4.4 hal_hmac_msp_deinit..... | 106 |
| 2.14.4.5 hal_hmac_sha256_digest..... | 106 |
| 2.14.4.6 hal_hmac_irq_handler..... | 107 |
| 2.14.4.7 hal_hmac_done_callback..... | 107 |
| 2.14.4.8 hal_hmac_error_callback..... | 107 |
| 2.14.4.9 hal_hmac_get_state..... | 108 |
| 2.14.4.10 hal_hmac_get_error..... | 108 |
| 2.14.4.11 hal_hmac_set_timeout..... | 108 |
| 2.14.4.12 hal_hmac_suspend_reg..... | 109 |
| 2.14.4.13 hal_hmac_resume_reg..... | 109 |
| 2.15 HAL PKC Generic Driver..... | 109 |
| 2.15.1 PKC Driver Functionalities..... | 109 |
| 2.15.2 How to Use PKC Driver..... | 110 |
| 2.15.3 PKC Driver Structures and Defines..... | 110 |
| 2.15.3.1 ecc_point_t..... | 110 |
| 2.15.3.2 ecc_curve_init_t..... | 111 |
| 2.15.3.3 pkc_init_t..... | 111 |
| 2.15.3.4 pkc_handle_t..... | 112 |
| 2.15.3.5 pkc_ecc_point_multi_t..... | 113 |
| 2.15.3.6 pkc_rsa_modular_exponent_t..... | 113 |
| 2.15.3.7 pkc_modular_add_t..... | 113 |
| 2.15.3.8 pkc_modular_sub_t..... | 113 |
| 2.15.3.9 pkc_modular_shift_t..... | 114 |
| 2.15.3.10 pkc_modular_compare_t..... | 114 |
| 2.15.3.11 pkc_montgomery_multi_t..... | 114 |
| 2.15.3.12 pkc_montgomery_inversion_t..... | 114 |
| 2.15.3.13 pkc_big_number_multi_t..... | 115 |
| 2.15.3.14 pkc_big_number_add_t..... | 115 |
| 2.15.4 PKC Driver APIs..... | 115 |

| | |
|---|-----|
| 2.15.4.1 hal_pkc_init..... | 116 |
| 2.15.4.2 hal_pkc_deinit..... | 117 |
| 2.15.4.3 hal_pkc_msp_init..... | 117 |
| 2.15.4.4 hal_pkc_msp_deinit..... | 117 |
| 2.15.4.5 hal_pkc_rsa_modular_exponent..... | 117 |
| 2.15.4.6 hal_pkc_ecc_point_multi..... | 118 |
| 2.15.4.7 hal_pkc_ecc_point_multi_it..... | 118 |
| 2.15.4.8 hal_pkc_modular_add..... | 118 |
| 2.15.4.9 hal_pkc_modular_add_it..... | 119 |
| 2.15.4.10 hal_pkc_modular_sub..... | 119 |
| 2.15.4.11 hal_pkc_modular_sub_it..... | 119 |
| 2.15.4.12 hal_pkc_modular_left_shift..... | 120 |
| 2.15.4.13 hal_pkc_modular_left_shift_it..... | 120 |
| 2.15.4.14 hal_pkc_modular_compare..... | 120 |
| 2.15.4.15 hal_pkc_modular_compare_it..... | 121 |
| 2.15.4.16 hal_pkc_montgomery_multi..... | 121 |
| 2.15.4.17 hal_pkc_montgomery_multi_it..... | 122 |
| 2.15.4.18 hal_pkc_montgomery_inversion..... | 122 |
| 2.15.4.19 hal_pkc_montgomery_inversion_it..... | 122 |
| 2.15.4.20 hal_pkc_big_number_multi..... | 123 |
| 2.15.4.21 hal_pkc_big_number_multi_it..... | 123 |
| 2.15.4.22 hal_pkc_big_number_add..... | 123 |
| 2.15.4.23 hal_pkc_big_number_add_it..... | 124 |
| 2.15.4.24 hal_pkc_irq_handler..... | 124 |
| 2.15.4.25 hal_pkc_done_callback..... | 124 |
| 2.15.4.26 hal_pkc_error_callback..... | 124 |
| 2.15.4.27 hal_pkc_overflow_callback..... | 125 |
| 2.15.4.28 hal_pkc_get_state..... | 125 |
| 2.15.4.29 hal_pkc_get_error..... | 125 |
| 2.15.4.30 hal_pkc_set_timeout..... | 126 |
| 2.15.4.31 hal_pkc_suspend_reg..... | 126 |
| 2.15.4.32 hal_pkc_resume_reg..... | 127 |
| 2.16 HAL I2C Generic Driver..... | 127 |
| 2.16.1 I2C Driver Functionalities..... | 127 |
| 2.16.2 How to Use I2C Driver..... | 127 |
| 2.16.2.1 I/O Read and Write in Polling Mode..... | 128 |
| 2.16.2.2 I/O Memory Read and Write in Polling Mode..... | 128 |
| 2.16.2.3 I/O Read and Write in Interrupt Mode..... | 128 |
| 2.16.2.4 I/O Memory Read and Write in Interrupt Mode..... | 129 |
| 2.16.2.5 I/O Read and Write in DMA Mode..... | 129 |
| 2.16.2.6 I/O Memory Read and Write in DMA Mode..... | 130 |
| 2.16.3 I2C Driver Structures..... | 130 |

| | |
|--|-----|
| 2.16.3.1 i2c_init_t..... | 130 |
| 2.16.3.2 i2c_handle_t..... | 131 |
| 2.16.4 I2C Driver APIs..... | 132 |
| 2.16.4.1 hal_i2c_init..... | 134 |
| 2.16.4.2 hal_i2c_deinit..... | 135 |
| 2.16.4.3 hal_i2c_msp_init..... | 135 |
| 2.16.4.4 hal_i2c_msp_deinit..... | 135 |
| 2.16.4.5 hal_i2c_master_transmit..... | 136 |
| 2.16.4.6 hal_i2c_master_receive..... | 136 |
| 2.16.4.7 hal_i2c_slave_transmit..... | 136 |
| 2.16.4.8 hal_i2c_slave_receive..... | 137 |
| 2.16.4.9 hal_i2c_mem_write..... | 137 |
| 2.16.4.10 hal_i2c_mem_read..... | 138 |
| 2.16.4.11 hal_i2c_master_transmit_it..... | 138 |
| 2.16.4.12 hal_i2c_master_receive_it..... | 139 |
| 2.16.4.13 hal_i2c_slave_transmit_it..... | 139 |
| 2.16.4.14 hal_i2c_slave_receive_it..... | 140 |
| 2.16.4.15 hal_i2c_mem_write_it..... | 140 |
| 2.16.4.16 hal_i2c_mem_read_it..... | 141 |
| 2.16.4.17 hal_i2c_master_abort_it..... | 142 |
| 2.16.4.18 hal_i2c_master_transmit_dma..... | 142 |
| 2.16.4.19 hal_i2c_master_receive_dma..... | 143 |
| 2.16.4.20 hal_i2c_slave_transmit_dma..... | 143 |
| 2.16.4.21 hal_i2c_slave_receive_dma..... | 144 |
| 2.16.4.22 hal_i2c_mem_write_dma..... | 144 |
| 2.16.4.23 hal_i2c_mem_read_dma..... | 145 |
| 2.16.4.24 hal_i2c_irq_handler..... | 145 |
| 2.16.4.25 hal_i2c_master_tx_cplt_callback..... | 146 |
| 2.16.4.26 hal_i2c_master_rx_cplt_callback..... | 146 |
| 2.16.4.27 hal_i2c_slave_tx_cplt_callback..... | 146 |
| 2.16.4.28 hal_i2c_slave_rx_cplt_callback..... | 147 |
| 2.16.4.29 hal_i2c_mem_tx_cplt_callback..... | 147 |
| 2.16.4.30 hal_i2c_mem_rx_cplt_callback..... | 147 |
| 2.16.4.31 hal_i2c_error_callback..... | 147 |
| 2.16.4.32 hal_i2c_abort_cplt_callback..... | 148 |
| 2.16.4.33 hal_i2c_get_state..... | 148 |
| 2.16.4.34 hal_i2c_get_mode..... | 149 |
| 2.16.4.35 hal_i2c_get_error..... | 149 |
| 2.16.4.36 hal_i2c_suspend_reg..... | 149 |
| 2.16.4.37 hal_i2c_resume_reg..... | 150 |
| 2.17 HAL QSPI Generic Driver..... | 150 |
| 2.17.1 QSPI Driver Functionalities..... | 150 |

| | |
|---|-----|
| 2.17.2 How to Use QSPI Driver..... | 151 |
| 2.17.3 QSPI Driver Structures..... | 151 |
| 2.17.3.1 qspi_init_t..... | 151 |
| 2.17.3.2 qspi_handle_t..... | 152 |
| 2.17.3.3 qspi_command_t..... | 154 |
| 2.17.4 QSPI Driver APIs..... | 156 |
| 2.17.4.1 hal_qspi_init..... | 157 |
| 2.17.4.2 hal_qspi_deinit..... | 157 |
| 2.17.4.3 hal_qspi_msp_init..... | 158 |
| 2.17.4.4 hal_qspi_msp_deinit..... | 158 |
| 2.17.4.5 hal_qspi_command_transmit..... | 158 |
| 2.17.4.6 hal_qspi_command_receive..... | 159 |
| 2.17.4.7 hal_qspi_command..... | 159 |
| 2.17.4.8 hal_qspi_transmit..... | 160 |
| 2.17.4.9 hal_qspi_receive..... | 160 |
| 2.17.4.10 hal_qspi_command_transmit_it..... | 160 |
| 2.17.4.11 hal_qspi_command_receive_it..... | 161 |
| 2.17.4.12 hal_qspi_command_it..... | 161 |
| 2.17.4.13 hal_qspi_transmit_it..... | 162 |
| 2.17.4.14 hal_qspi_receive_it..... | 162 |
| 2.17.4.15 hal_qspi_command_transmit_dma..... | 162 |
| 2.17.4.16 hal_qspi_command_receive_dma..... | 163 |
| 2.17.4.17 hal_qspi_command_dma..... | 163 |
| 2.17.4.18 hal_qspi_transmit_dma..... | 164 |
| 2.17.4.19 hal_qspi_receive_dma..... | 164 |
| 2.17.4.20 hal_qspi_abort..... | 165 |
| 2.17.4.21 hal_qspi_abort_it..... | 165 |
| 2.17.4.22 hal_qspi_irq_handler..... | 165 |
| 2.17.4.23 hal_qspi_tx_cplt_callback..... | 165 |
| 2.17.4.24 hal_qspi_rx_cplt_callback..... | 166 |
| 2.17.4.25 hal_qspi_error_callback..... | 166 |
| 2.17.4.26 hal_qspi_abort_cplt_callback..... | 166 |
| 2.17.4.27 hal_qspi_get_state..... | 167 |
| 2.17.4.28 hal_qspi_get_error..... | 167 |
| 2.17.4.29 hal_qspi_set_timeout..... | 167 |
| 2.17.4.30 hal_qspi_set_tx_fifo_threshold..... | 168 |
| 2.17.4.31 hal_qspi_set_rx_fifo_threshold..... | 168 |
| 2.17.4.32 hal_qspi_get_tx_fifo_threshold..... | 168 |
| 2.17.4.33 hal_qspi_get_rx_fifo_threshold..... | 169 |
| 2.17.4.34 hal_qspi_suspend_reg..... | 169 |
| 2.17.4.35 hal_qspi_resume_reg..... | 169 |
| 2.18 HAL PWM Generic Driver..... | 170 |

| | |
|---|-----|
| 2.18.1 PWM Driver Functionalities..... | 170 |
| 2.18.2 How to Use PWM Driver..... | 170 |
| 2.18.3 PWM Driver Structures..... | 170 |
| 2.18.3.1 pwm_channel_init_t..... | 171 |
| 2.18.3.2 pwm_init_t..... | 171 |
| 2.18.3.3 pwm_handle_t..... | 172 |
| 2.18.4 PWM Driver APIs..... | 173 |
| 2.18.4.1 hal_pwm_init..... | 173 |
| 2.18.4.2 hal_pwm_deinit..... | 173 |
| 2.18.4.3 hal_pwm_msp_init..... | 174 |
| 2.18.4.4 hal_pwm_msp_deinit..... | 174 |
| 2.18.4.5 hal_pwm_start..... | 174 |
| 2.18.4.6 hal_pwm_stop..... | 174 |
| 2.18.4.7 hal_pwm_update_freq..... | 175 |
| 2.18.4.8 hal_pwm_config_channel..... | 175 |
| 2.18.4.9 hal_pwm_get_state..... | 175 |
| 2.18.4.10 hal_pwm_suspend_reg..... | 176 |
| 2.18.4.11 hal_pwm_resume_reg..... | 176 |
| 2.19 HAL PWR Generic Driver..... | 177 |
| 2.19.1 PWR Driver Functionalities..... | 177 |
| 2.19.2 How to Use PWR Driver..... | 177 |
| 2.19.2.1 Bluetooth LE Power Configuration..... | 177 |
| 2.19.2.2 Ultra Deep Sleep Configuration..... | 177 |
| 2.19.3 PWR Driver APIs..... | 178 |
| 2.19.3.1 hal_pwr_set_wakeup_condition..... | 178 |
| 2.19.3.2 hal_pwr_config_timer_wakeup..... | 179 |
| 2.19.3.3 hal_pwr_config_ext_wakeup..... | 179 |
| 2.19.3.4 hal_pwr_set_comm_power..... | 180 |
| 2.19.3.5 hal_pwr_set_comm_mode..... | 180 |
| 2.19.3.6 hal_pwr_enter_chip_deepsleep..... | 181 |
| 2.19.3.7 hal_pwr_get_timer_current_value..... | 181 |
| 2.19.3.8 hal_pwr_disable_ext_wakeup..... | 181 |
| 2.19.3.9 hal_pwr_sleep_timer_irq_handler..... | 182 |
| 2.19.3.10 hal_pwr_sleep_timer_elapsed_callback..... | 182 |
| 2.20 HAL SPI Generic Driver..... | 182 |
| 2.20.1 SPI Driver Functionalities..... | 183 |
| 2.20.2 How to Use SPI Driver..... | 183 |
| 2.20.3 SPI Driver Structures..... | 184 |
| 2.20.3.1 spi_init_t..... | 184 |
| 2.20.3.2 spi_handle_t..... | 185 |
| 2.20.4 SPI Driver APIs..... | 187 |
| 2.20.4.1 hal_spi_init..... | 189 |

| | |
|--|-----|
| 2.20.4.2 hal_spi_deinit..... | 189 |
| 2.20.4.3 hal_spi_msp_init..... | 189 |
| 2.20.4.4 hal_spi_msp_deinit..... | 190 |
| 2.20.4.5 hal_spi_transmit..... | 190 |
| 2.20.4.6 hal_spi_receive..... | 190 |
| 2.20.4.7 hal_spi_transmit_receive..... | 191 |
| 2.20.4.8 hal_spi_read_eeprom..... | 191 |
| 2.20.4.9 hal_spi_transmit_it..... | 192 |
| 2.20.4.10 hal_spi_receive_it..... | 192 |
| 2.20.4.11 hal_spi_transmit_receive_it..... | 192 |
| 2.20.4.12 hal_spi_read_eeprom_it..... | 193 |
| 2.20.4.13 hal_spi_transmit_dma..... | 194 |
| 2.20.4.14 hal_spi_receive_dma..... | 194 |
| 2.20.4.15 hal_spi_transmit_receive_dma..... | 194 |
| 2.20.4.16 hal_spi_read_eeprom_dma..... | 195 |
| 2.20.4.17 hal_spi_abort..... | 195 |
| 2.20.4.18 hal_spi_abort_it..... | 196 |
| 2.20.4.19 hal_spi_irq_handler..... | 196 |
| 2.20.4.20 hal_spi_tx_cplt_callback..... | 196 |
| 2.20.4.21 hal_spi_rx_cplt_callback..... | 197 |
| 2.20.4.22 hal_spi_tx_rx_cplt_callback..... | 197 |
| 2.20.4.23 hal_spi_error_callback..... | 197 |
| 2.20.4.24 hal_spi_abort_cplt_callback..... | 198 |
| 2.20.4.25 hal_spi_get_state..... | 198 |
| 2.20.4.26 hal_spi_get_error..... | 198 |
| 2.20.4.27 hal_spi_set_timeout..... | 199 |
| 2.20.4.28 hal_spi_set_tx_fifo_threshold..... | 199 |
| 2.20.4.29 hal_spi_set_rx_fifo_threshold..... | 199 |
| 2.20.4.30 hal_spi_get_tx_fifo_threshold..... | 200 |
| 2.20.4.31 hal_spi_get_rx_fifo_threshold..... | 200 |
| 2.20.4.32 hal_spi_suspend_reg..... | 200 |
| 2.20.4.33 hal_spi_resume_reg..... | 200 |
| 2.21 HAL TIMER Generic Driver..... | 201 |
| 2.21.1 TIMER Driver Functionalities..... | 201 |
| 2.21.2 How to Use TIMER Driver..... | 201 |
| 2.21.3 TIMER Driver Structures..... | 202 |
| 2.21.3.1 timer_init_t..... | 202 |
| 2.21.3.2 timer_handle_t..... | 202 |
| 2.21.4 TIMER Driver APIs..... | 202 |
| 2.21.4.1 hal_timer_base_init..... | 203 |
| 2.21.4.2 hal_timer_base_deinit..... | 203 |
| 2.21.4.3 hal_timer_base_msp_init..... | 203 |

| | |
|--|-----|
| 2.21.4.4 hal_timer_base_msp_deinit..... | 204 |
| 2.21.4.5 hal_timer_base_start..... | 204 |
| 2.21.4.6 hal_timer_base_stop..... | 204 |
| 2.21.4.7 hal_timer_base_start_it..... | 205 |
| 2.21.4.8 hal_timer_base_stop_it..... | 205 |
| 2.21.4.9 hal_timer_set_config..... | 205 |
| 2.21.4.10 hal_timer_irq_handler..... | 205 |
| 2.21.4.11 hal_timer_period_elapsed_callback..... | 206 |
| 2.21.4.12 hal_timer_get_state..... | 206 |
| 2.22 HAL Calendar Generic Driver..... | 206 |
| 2.22.1 Calendar Driver Functionalities..... | 206 |
| 2.22.2 How to Use Calendar Driver..... | 207 |
| 2.22.3 Calendar Driver Structures..... | 207 |
| 2.22.3.1 calendar_time_t..... | 207 |
| 2.22.3.2 calendar_alarm_t..... | 207 |
| 2.22.3.3 calendar_handle_t..... | 208 |
| 2.22.4 Calendar Driver APIs..... | 209 |
| 2.22.4.1 hal_calendar_init..... | 209 |
| 2.22.4.2 hal_calendar_deinit..... | 209 |
| 2.22.4.3 hal_calendar_init_time..... | 210 |
| 2.22.4.4 hal_calendar_get_time..... | 210 |
| 2.22.4.5 hal_calendar_set_alarm..... | 210 |
| 2.22.4.6 hal_calendar_set_tick..... | 211 |
| 2.22.4.7 hal_calendar_disable_event..... | 211 |
| 2.22.4.8 hal_calendar_irq_handler..... | 211 |
| 2.22.4.9 hal_calendar_alarm_callback..... | 211 |
| 2.22.4.10 hal_calendar_tick_callback..... | 212 |
| 2.23 HAL UART Generic Driver..... | 212 |
| 2.23.1 UART Driver Functionalities..... | 212 |
| 2.23.2 How to Use UART Driver..... | 213 |
| 2.23.3 UART Driver Structures..... | 213 |
| 2.23.3.1 uart_init_t..... | 213 |
| 2.23.3.2 uart_handle_t..... | 214 |
| 2.23.4 UART Driver APIs..... | 216 |
| 2.23.4.1 hal_uart_init..... | 217 |
| 2.23.4.2 hal_uart_deinit..... | 218 |
| 2.23.4.3 hal_uart_msp_init..... | 218 |
| 2.23.4.4 hal_uart_msp_deinit..... | 218 |
| 2.23.4.5 hal_uart_transmit..... | 219 |
| 2.23.4.6 hal_uart_receive..... | 219 |
| 2.23.4.7 hal_uart_transmit_it..... | 219 |
| 2.23.4.8 hal_uart_receive_it..... | 220 |

| | |
|--|-----|
| 2.23.4.9 hal_uart_transmit_dma..... | 220 |
| 2.23.4.10 hal_uart_receive_dma..... | 220 |
| 2.23.4.11 hal_uart_dma_pause..... | 220 |
| 2.23.4.12 hal_uart_dma_resume..... | 221 |
| 2.23.4.13 hal_uart_dma_stop..... | 221 |
| 2.23.4.14 hal_uart_abort..... | 221 |
| 2.23.4.15 hal_uart_abort_transmit..... | 222 |
| 2.23.4.16 hal_uart_abort_receive..... | 222 |
| 2.23.4.17 hal_uart_abort_it..... | 222 |
| 2.23.4.18 hal_uart_abort_transmit_it..... | 223 |
| 2.23.4.19 hal_uart_abort_receive_it..... | 223 |
| 2.23.4.20 hal_uart_irq_handler..... | 224 |
| 2.23.4.21 hal_uart_tx_cplt_callback..... | 224 |
| 2.23.4.22 hal_uart_rx_cplt_callback..... | 224 |
| 2.23.4.23 hal_uart_error_callback..... | 224 |
| 2.23.4.24 hal_uart_abort_cplt_callback..... | 225 |
| 2.23.4.25 hal_uart_abort_tx_cplt_callback..... | 225 |
| 2.23.4.26 hal_uart_abort_rx_cplt_callback..... | 225 |
| 2.23.4.27 hal_uart_get_state..... | 226 |
| 2.23.4.28 hal_uart_get_error..... | 226 |
| 2.23.4.29 hal_uart_suspend_reg..... | 227 |
| 2.23.4.30 hal_uart_resume_reg..... | 227 |
| 2.24 HAL I2S Generic Driver..... | 227 |
| 2.24.1 I2S Driver Functionalities..... | 227 |
| 2.24.2 How to Use I2S Driver..... | 228 |
| 2.24.2.1 I/O Read and Write in Polling Mode..... | 228 |
| 2.24.2.2 I/O Read and Write in Interrupt Mode..... | 228 |
| 2.24.2.3 I/O Read and Write in DMA Mode..... | 229 |
| 2.24.3 I2S Driver Structures..... | 229 |
| 2.24.3.1 i2s_init_t..... | 229 |
| 2.24.3.2 i2s_handle_t..... | 230 |
| 2.24.4 I2S Driver APIs..... | 232 |
| 2.24.4.1 hal_i2s_init..... | 233 |
| 2.24.4.2 hal_i2s_deinit..... | 233 |
| 2.24.4.3 hal_i2s_msp_init..... | 233 |
| 2.24.4.4 hal_i2s_msp_deinit..... | 234 |
| 2.24.4.5 hal_i2s_transmit..... | 234 |
| 2.24.4.6 hal_i2s_receive..... | 234 |
| 2.24.4.7 hal_i2s_transmit_receive..... | 235 |
| 2.24.4.8 hal_i2s_transmit_it..... | 235 |
| 2.24.4.9 hal_i2s_receive_it..... | 235 |
| 2.24.4.10 hal_i2s_transmit_receive_it..... | 236 |

| | |
|---|-----|
| 2.24.4.11 hal_i2s_abort..... | 236 |
| 2.24.4.12 hal_i2s_transmit_dma..... | 237 |
| 2.24.4.13 hal_i2s_receive_dma()..... | 237 |
| 2.24.4.14 hal_i2s_transmit_receive_dma..... | 237 |
| 2.24.4.15 hal_i2s_irq_handler..... | 238 |
| 2.24.4.16 hal_i2s_tx_cplt_callback..... | 238 |
| 2.24.4.17 hal_i2s_tx_rx_cplt_callback..... | 238 |
| 2.24.4.18 hal_i2s_rx_cplt_callback..... | 239 |
| 2.24.4.19 hal_i2s_error_callback..... | 239 |
| 2.24.4.20 hal_i2s_get_state..... | 239 |
| 2.24.4.21 hal_i2s_get_error..... | 240 |
| 2.24.4.22 hal_i2s_start_clock..... | 240 |
| 2.24.4.23 hal_i2s_stop_clock..... | 240 |
| 2.24.4.24 hal_i2s_set_tx_fifo_threshold..... | 241 |
| 2.24.4.25 hal_i2s_set_rx_fifo_threshold..... | 241 |
| 2.24.4.26 hal_i2s_get_tx_fifo_threshold..... | 241 |
| 2.24.4.27 hal_i2s_get_rx_fifo_threshold..... | 242 |
| 2.24.4.28 hal_i2s_suspend_reg..... | 242 |
| 2.24.4.29 hal_i2s_resume_reg..... | 242 |
| 2.25 HAL RNG Generic Driver..... | 242 |
| 2.25.1 RNG Driver Functionalities..... | 242 |
| 2.25.2 How to Use RNG Driver..... | 243 |
| 2.25.3 RNG Driver Structures..... | 243 |
| 2.25.3.1 rng_init_t..... | 243 |
| 2.25.3.2 rng_handle_t..... | 244 |
| 2.25.4 RNG Driver APIs..... | 245 |
| 2.25.4.1 hal_rng_init..... | 245 |
| 2.25.4.2 hal_rng_deinit..... | 245 |
| 2.25.4.3 hal_rng_msp_init..... | 246 |
| 2.25.4.4 hal_rng_msp_deinit..... | 246 |
| 2.25.4.5 hal_rng_generate_random_number..... | 246 |
| 2.25.4.6 hal_rng_generate_random_number_it..... | 247 |
| 2.25.4.7 hal_rng_read_last_random_number..... | 247 |
| 2.25.4.8 hal_rng_irq_handler..... | 247 |
| 2.25.4.9 hal_rng_ready_data_callback..... | 248 |
| 2.25.4.10 hal_rng_suspend_reg..... | 248 |
| 2.25.4.11 hal_rng_resume_reg..... | 248 |
| 2.26 HAL AON WDT Generic Driver..... | 248 |
| 2.26.1 AON WDT Driver Functionalities..... | 248 |
| 2.26.2 How to Use AON WDT Driver..... | 249 |
| 2.26.3 AON WDT Driver Structures..... | 249 |
| 2.26.3.1 aon_wdt_init_t..... | 249 |

| | |
|---|------------|
| 2.26.3.2 aon_wdt_handle_t..... | 249 |
| 2.26.4 AON WDT Driver APIs..... | 249 |
| 2.26.4.1 hal_aon_wdt_init..... | 250 |
| 2.26.4.2 hal_aon_wdt_deinit..... | 250 |
| 2.26.4.3 hal_aon_wdt_refresh..... | 250 |
| 2.26.4.4 hal_aon_wdt_irq_handler..... | 251 |
| 2.26.4.5 hal_aon_wdt_alarm_callback..... | 251 |
| 2.27 HAL WDT Generic Driver..... | 251 |
| 2.27.1 WDT Driver Functionalities..... | 251 |
| 2.27.2 How to Use WDT Driver..... | 251 |
| 2.27.3 WDT Driver Structures..... | 252 |
| 2.27.3.1 wdt_init_t..... | 252 |
| 2.27.3.2 wdt_handle_t..... | 252 |
| 2.27.4 WDT Driver APIs..... | 252 |
| 2.27.4.1 hal_wdt_init..... | 253 |
| 2.27.4.2 hal_wdt_deinit..... | 253 |
| 2.27.4.3 hal_wdt_msp_init..... | 253 |
| 2.27.4.4 hal_wdt_msp_deinit..... | 254 |
| 2.27.4.5 hal_wdt_refresh..... | 254 |
| 2.27.4.6 hal_wdt_irq_handler..... | 254 |
| 2.27.4.7 hal_wdt_period_elapsed_callback..... | 255 |
| 2.28 HAL COMP Generic Driver..... | 255 |
| 2.28.1 COMP Driver Functionalities..... | 255 |
| 2.28.2 How to Use COMP Driver..... | 255 |
| 2.28.3 COMP Driver Structures..... | 256 |
| 2.28.3.1 comp_init_t..... | 256 |
| 2.28.3.2 comp_handle_t..... | 256 |
| 2.28.4 COMP Driver APIs..... | 256 |
| 2.28.4.1 hal_comp_init..... | 257 |
| 2.28.4.2 hal_comp_deinit..... | 257 |
| 2.28.4.3 hal_comp_msp_init..... | 258 |
| 2.28.4.4 hal_comp_msp_deinit()..... | 258 |
| 2.28.4.5 hal_comp_start..... | 258 |
| 2.28.4.6 hal_comp_stop..... | 259 |
| 2.28.4.7 hal_comp_irq_handler..... | 259 |
| 2.28.4.8 hal_comp_trigger_callback..... | 259 |
| 2.28.4.9 hal_comp_get_state..... | 259 |
| 2.28.4.10 hal_comp_get_error..... | 260 |
| 2.28.4.11 hal_comp_suspend_reg..... | 260 |
| 2.28.4.12 hal_comp_resume_reg..... | 260 |
| 3 LL Drivers..... | 262 |
| 3.1 Introduction..... | 262 |

| | |
|--|-----|
| 3.1.1 LL Common Resources..... | 262 |
| 3.1.2 How to Use LL Drivers..... | 262 |
| 3.2 LL GPIO Generic Driver..... | 263 |
| 3.2.1 GPIO Driver Structure..... | 263 |
| 3.2.1.1 ll_gpio_init_t..... | 263 |
| 3.2.2 GPIO Driver APIs..... | 264 |
| 3.2.2.1 ll_gpio_init..... | 264 |
| 3.2.2.2 ll_gpio_deinit..... | 265 |
| 3.2.2.3 ll_gpio_struct_init..... | 265 |
| 3.3 LL AON GPIO Generic Driver..... | 265 |
| 3.3.1 AON GPIO Driver Structure..... | 265 |
| 3.3.1.1 ll_aon_gpio_init_t..... | 265 |
| 3.3.2 AON GPIO Driver APIs..... | 266 |
| 3.3.2.1 ll_aon_gpio_init..... | 267 |
| 3.3.2.2 ll_aon_gpio_deinit..... | 267 |
| 3.3.2.3 ll_aon_gpio_struct_init..... | 267 |
| 3.4 LL ADC Generic Driver..... | 268 |
| 3.4.1 ADC Driver Structure..... | 268 |
| 3.4.1.1 ll_adc_init_t..... | 268 |
| 3.4.2 ADC Driver APIs..... | 269 |
| 3.4.2.1 ll_adc_init..... | 269 |
| 3.4.2.2 ll_adc_deinit..... | 270 |
| 3.4.2.3 ll_adc_struct_init..... | 270 |
| 3.5 LL DMA Generic Driver..... | 270 |
| 3.5.1 DMA Driver Structure..... | 270 |
| 3.5.1.1 ll_dma_init_t..... | 270 |
| 3.5.2 DMA Driver APIs..... | 273 |
| 3.5.2.1 ll_dma_init..... | 274 |
| 3.5.2.2 ll_dma_deinit..... | 274 |
| 3.5.2.3 ll_dma_struct_init..... | 275 |
| 3.6 LL DUAL TIMER Generic Driver..... | 275 |
| 3.6.1 DUAL TIMER Driver Structure..... | 275 |
| 3.6.1.1 ll_dual_timer_init_t..... | 275 |
| 3.6.2 DUAL TIMER Driver APIs..... | 276 |
| 3.6.2.1 ll_dual_timer_init..... | 276 |
| 3.6.2.2 ll_dual_timer_deinit..... | 276 |
| 3.6.2.3 ll_dual_timer_struct_init..... | 277 |
| 3.7 LL I2C Generic Driver..... | 277 |
| 3.7.1 I2C Driver Structures..... | 277 |
| 3.7.1.1 ll_i2c_init_t..... | 277 |
| 3.7.2 I2C Driver APIs..... | 277 |
| 3.7.2.1 ll_i2c_init..... | 278 |

| | |
|-------------------------------------|-----|
| 3.7.2.2 ll_i2c_deinit..... | 278 |
| 3.7.2.3 ll_i2c_struct_init..... | 278 |
| 3.8 LL MSIO Generic Driver..... | 279 |
| 3.8.1 MSIO Driver Structure..... | 279 |
| 3.8.1.1 ll_msio_init_t..... | 279 |
| 3.8.2 MSIO Driver APIs..... | 280 |
| 3.8.2.1 ll_msio_init..... | 280 |
| 3.8.2.2 ll_msio_deinit..... | 280 |
| 3.8.2.3 ll_msio_struct_init..... | 280 |
| 3.9 LL AES Generic Driver..... | 281 |
| 3.9.1 AES Driver Structure..... | 281 |
| 3.9.1.1 ll_aes_init_t..... | 281 |
| 3.9.2 AES Driver APIs..... | 281 |
| 3.9.2.1 ll_aes_init..... | 281 |
| 3.9.2.2 ll_aes_deinit..... | 282 |
| 3.9.2.3 ll_aes_struct_init..... | 282 |
| 3.10 LL PKC Generic Driver..... | 282 |
| 3.10.1 PKC Driver Structures..... | 282 |
| 3.10.1.1 ll_ecc_point_t..... | 282 |
| 3.10.1.2 ll_ecc_curve_init_t..... | 283 |
| 3.10.1.3 ll_pkc_init_t..... | 283 |
| 3.10.2 PKC Driver APIs..... | 283 |
| 3.10.2.1 ll_pkc_init..... | 284 |
| 3.10.2.2 ll_pkc_deinit..... | 284 |
| 3.10.2.3 ll_pkc_struct_init..... | 284 |
| 3.11 LL PWM Generic Driver..... | 285 |
| 3.11.1 PWM Driver Structures..... | 285 |
| 3.11.1.1 ll_pwm_channel_init_t..... | 285 |
| 3.11.1.2 ll_pwm_init_t..... | 285 |
| 3.11.2 PWM Driver APIs..... | 286 |
| 3.11.2.1 ll_pwm_init..... | 286 |
| 3.11.2.2 ll_pwm_deinit..... | 286 |
| 3.11.2.3 ll_pwm_struct_init..... | 287 |
| 3.12 LL SPI Generic Driver..... | 287 |
| 3.12.1 SPI Driver Structures..... | 287 |
| 3.12.1.1 ll_spim_init_t..... | 287 |
| 3.12.1.2 ll_spis_init_t..... | 289 |
| 3.12.1.3 ll_qspi_init_t..... | 290 |
| 3.12.2 SPI Driver APIs..... | 292 |
| 3.12.2.1 ll_spim_init..... | 293 |
| 3.12.2.2 ll_spim_deinit..... | 293 |
| 3.12.2.3 ll_spim_struct_init..... | 293 |

| | |
|--|------------|
| 3.12.2.4 ll_spis_init..... | 294 |
| 3.12.2.5 ll_spis_deinit..... | 294 |
| 3.12.2.6 ll_spis_struct_init..... | 294 |
| 3.12.2.7 ll_qspi_init..... | 295 |
| 3.12.2.8 ll_qspi_deinit..... | 295 |
| 3.12.2.9 ll_qspi_struct_init..... | 295 |
| 3.13 LL TIMER Generic Driver..... | 295 |
| 3.13.1 TIMER Driver Structure..... | 295 |
| 3.13.1.1 ll_timer_init_t..... | 296 |
| 3.13.2 TIMER Driver APIs..... | 296 |
| 3.13.2.1 ll_timer_init..... | 296 |
| 3.13.2.2 ll_timer_deinit..... | 296 |
| 3.13.2.3 ll_timer_struct_init..... | 297 |
| 3.14 LL UART Generic Driver..... | 297 |
| 3.14.1 UART Driver Structure..... | 297 |
| 3.14.1.1 ll_uart_init_t..... | 297 |
| 3.14.2 UART Driver APIs..... | 298 |
| 3.14.2.1 ll_uart_init..... | 298 |
| 3.14.2.2 ll_uart_deinit..... | 298 |
| 3.14.2.3 ll_uart_struct_init..... | 299 |
| 3.15 LL I2S Generic Driver..... | 299 |
| 3.15.1 I2S Driver Structure..... | 299 |
| 3.15.1.1 ll_i2s_init_t..... | 299 |
| 3.15.2 I2S Driver APIs..... | 301 |
| 3.15.2.1 ll_i2s_init..... | 301 |
| 3.15.2.2 ll_i2s_deinit..... | 302 |
| 3.15.2.3 ll_i2s_struct_init..... | 302 |
| 3.16 LL RNG Generic Driver..... | 302 |
| 3.16.1 RNG Driver Structure..... | 302 |
| 3.16.1.1 ll_rng_init_t..... | 302 |
| 3.16.2 RNG Driver APIs..... | 303 |
| 3.16.2.1 ll_rng_init..... | 304 |
| 3.16.2.2 ll_rng_deinit..... | 304 |
| 3.16.2.3 ll_rng_struct_init..... | 304 |
| 3.17 LL COMP Generic Driver..... | 305 |
| 3.17.1 COMP Driver Structures..... | 305 |
| 3.17.1.1 ll_comp_init_t..... | 305 |
| 3.17.2 COMP Driver APIs..... | 305 |
| 3.17.2.1 ll_comp_init..... | 306 |
| 3.17.2.2 ll_comp_deinit..... | 306 |
| 3.17.2.3 ll_comp_struct_init..... | 306 |
| 4 Glossary and Abbreviations..... | 307 |

1 Overview

1.1 GR551x Peripheral Drivers

The GR551x peripheral drivers comprise Hardware Abstraction Layer (HAL) drivers and Low Layer (LL) drivers. The driver architecture is shown in [Figure 1-1](#).

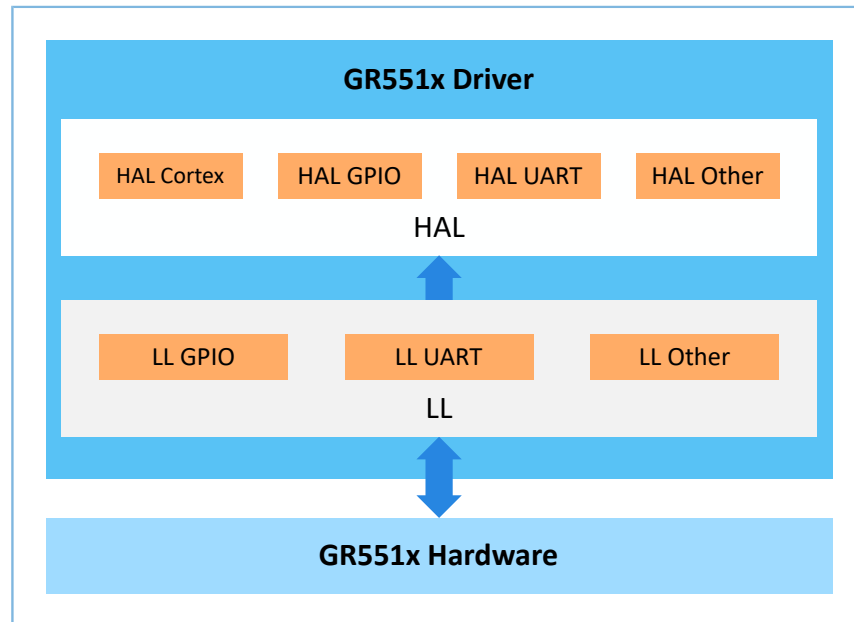


Figure 1-1 GR551x driver architecture

At the software layer, the HAL is in close association with the LL. LL APIs are called when internal HAL registers are accessed. In general, developers need to use HAL APIs to implement corresponding functions. For some special functions that cannot be implemented by using HAL APIs, developers need to call LL APIs to quickly encapsulate the required APIs.

1.1.1 HAL Drivers

The HAL drivers encapsulate most-commonly-used functions of all peripherals into a set of easy-to-use APIs, allowing developers to implement interactions between LL peripherals and upper-layer applications.

The HAL drivers provide the following features:

- Intra-series APIs covering the common SoC functions as well as extension APIs for special SoC functions
- Three API programming modes: polling, interrupt, and Direct Memory Access (DMA)
- Fully reentrant and RTOS-compliant APIs
- Support of multi-instance, allowing concurrent API calls for multiple instances of a peripheral (such as I2C0 and I2C1 in I2C)
- Call for user callback functions in initialization/deinitialization APIs to initialize or deinitialize General Purpose Input/Output (GPIO), interrupt, and DMA

- Call for callback functions in peripheral interrupt and error events to inform users that certain events have been triggered
- Support of locking mechanism, enabling safe access to shared resources
- Timeout for polling operations to prevent an infinite loop

1.1.2 LL Drivers

The LL drivers encapsulate atomic operations of all peripheral registers by using inline functions. The LL drivers which are closer to hardware than the HAL drivers offer API functions covering all peripheral features. Developers can use LL drivers to configure peripheral features that are not covered by the HAL drivers. For performance-demanding scenarios or those with limited storage space, developers can directly use LL drivers.

The LL drivers provide the following features:

- Support of inline functions to prevent function call overhead
- Encapsulation of register operations in LL drivers with a high-level portability and ease-of-reuse
- Offering a wide range of features functionalities

1.2 File Classification

The HAL and LL files are classified into two categories: driver files and user-application files.

- Driver files: header files, HAL driver files, and LL driver files
- User-application files: files that are referred to or implemented when users build a project

1.2.1 Driver Files

1.2.1.1 Header Files

Table 1-1 Header files

| Name | Description |
|-----------|--|
| gr55xx.h | Header file that is common to all GR55xx SoCs series. It contains header file description, some public macro statements, and enumeration declarations. Examples: <code>flag_status_t</code> , <code>SET_BIT(REG, BIT)</code> , and <code>gr55xx_hal.h</code> . |
| gr551xx.h | Processor header file that is common to all GR551x SoCs series. It contains structure declarations of all peripheral control registers as well as peripherals. The declarations are implemented by using structures and macros. |

1.2.1.2 HAL Driver Files

The components of the HAL driver files in a GR551x SoC are listed in [Figure 1-2](#).

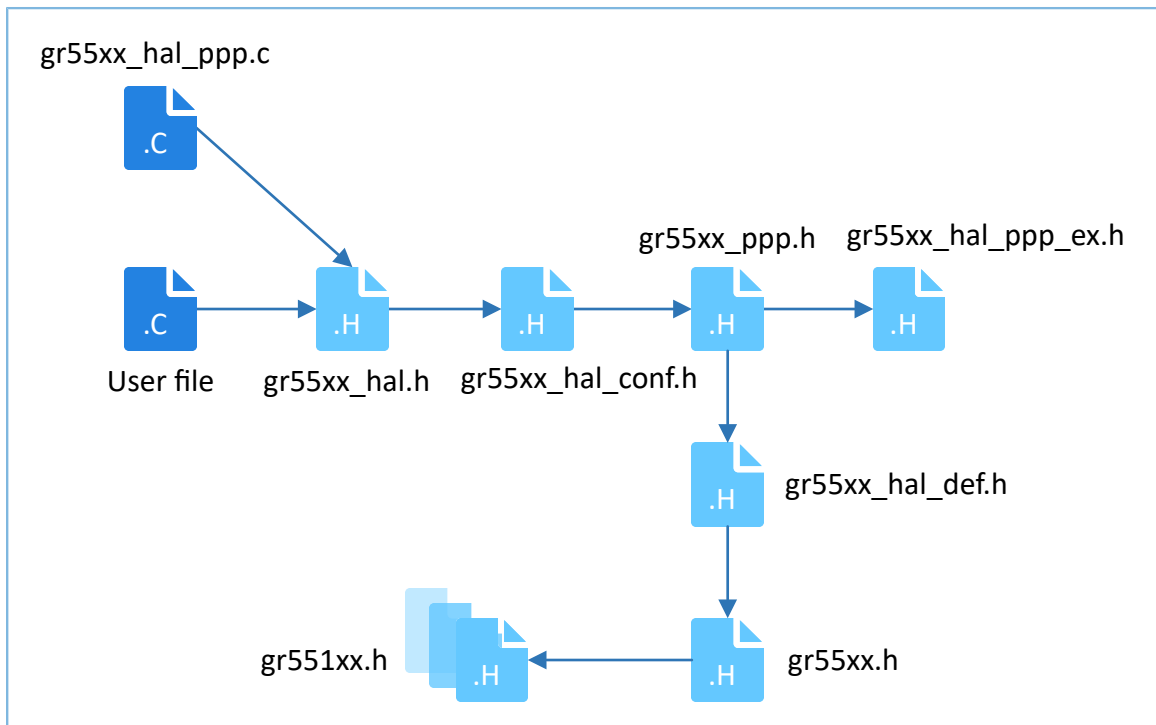


Figure 1-2 HAL driver files

The table below lists the description of HAL driver files.

Table 1-2 Description of HAL driver files

| Name | Description |
|---------------------|--|
| gr55xx_hal_conf.h | Configuration file for HAL drivers. It contains HAL driver header files for all peripherals. Developers can modify this file to specify a peripheral/module and other parameters to be compiled. |
| gr55xx_hal_ppp.h | HAL driver header file of a module. It contains API functions, structures, enumerations, and macros. Example: <i>gr55xx_hal_uart.h</i> |
| gr55xx_hal_ppp.c | HAL driver source file of a module. It helps implement driver API functions. Example: <i>gr55xx_hal_uart.c</i> |
| gr55xx_hal_ppp_ex.h | Header file of extension features for some module drivers. It contains function declarations of the extension features that are available on some SoCs. Example: <i>gr55xx_hal_gpio_ex.h</i> that contains defines statements of pin multiplexing |
| gr55xx_hal_ppp_ex.c | Source file of extension features for some module drivers. To date, the functions in this file are not available. |
| gr55xx_hal.h | Header file that is common to HAL drivers. It contains configuration header file as well statements of hal_init and related tick API functions. By introducing this file into applications, developers can use HAL drivers of GR551x SoCs. This file bridges user applications to HAL drivers. |

| Name | Description |
|---------------------------|---|
| gr55xx_hal.c | Source file that is common to HAL drivers. It helps implement hal_init() and related tick() API functions (weak function, can be re-defined on demand). |
| gr55xx_hal_msp_template.c | Template file allowing implementation of hal_ppp_msp_init() and hal_ppp_msp_deinit() API functions. It configures MspInit and MspDeinit callback functions of all peripherals in a unified manner. Among them, hal_ppp_msp_init() is called in hal_ppp_init() to configure GPIO reuse, clock, DMA, and interrupts of corresponding modules. |
| gr55xx_hal_def.h | Type define file for HAL drivers. It contains common macros, structures, and enumeration declarations as well as compiler-related define statements. Example: hal_status_t. Declaration header file that is common to HAL drivers. It contains common data types and constants of all peripheral drivers. |

 **Note:**

ppp represents the peripheral name. Examples: gpio, qspi, and uart

1.2.1.3 LL Driver Files

The table below lists the description of LL driver files.

Table 1-3 Description of LL driver files

| Name | Description |
|-----------------|---|
| gr55xx_ll_ppp.h | LL driver header file of a module. It contains macro definitions and structure declarations of LL drivers of the module. It also helps implement inline functions for register access in LL drivers. Example: <i>gr55xx_ll_gpio.h</i> |
| gr55xx_ll_ppp.c | LL driver source file of a module. It contains init() and deinit() functions. Example: <i>gr55xx_ll_gpio.c</i> |

1.2.2 User-Application Files

The table below lists the user-application files of GR551x SoCs.

Table 1-4 Description of user-application files

| Name | Description |
|------------------|--|
| system_gr55xx.c | This file contains the SystemInit() function to perform system initialization after system reset. The file also helps configure system clocks. |
| startup_gr55xx.s | Startup file for GR551x SoCs |
| gr55xx_hal_msp.c | Optional file. This file contains msp_init() and msp_deinit() functions of all peripheral modules. Those of peripherals that are less frequently used can be stored in <i>main.c</i> . |

| Name | Description |
|----------------|--|
| gr55xx_it.c/.h | Optional file. This file contains interrupt handlers of peripherals. Those of peripherals that are less frequently used can be stored in <i>main.c</i> . |
| main.c/.h | It contains main() functions and header files. |

1.3 API Classification

The HAL and LL driver APIs are categorized into two groups: generic APIs and extension APIs.

1.3.1 Generic APIs

Generic APIs offer common and generic functions for all GR551x SoCs series. Differences between HAL generic APIs and LL generic APIs are elaborated in the sections below.

1.3.1.1 HAL Generic APIs

Based on API roles, the HAL generic APIs are categorized into five types:

- Initialization type: These APIs initialize/deinitialize peripherals and peripheral-specific public system resources. They help perform pull up/down for GPIO pins and functionality multiplexing, enable Nested Vectored Interrupt Controller (NVIC) interrupts, and initialize DMA channels. Example: `hal_uart_init()`.
- I/O operation type: These APIs allow data transmission and reception for UART. Example: `hal_uart_transmit()`.
- Interrupt handling and callback function type: These APIs are used to handle interrupts and call callback functions of peripherals. Examples: `hal_uart_irq_handler()` and `hal_uart_tx_cplt_callback()`.
- Control type: These APIs are used to set feature parameters of peripherals. Example: `hal_spi_set_tx_fifo_threshold()`.
- State and error type: These APIs are used to retrieve operating state and error code of HAL drivers. Example: `hal_i2c_get_state()`.

1.3.1.2 LL Generic APIs

Based on API roles, the LL generic APIs are categorized into six types:

- Initialization type: These APIs initialize/deinitialize peripherals. Example: `ll_pwm_init()`.
- Feature enablement type: These APIs enable/disable certain peripheral features. Example: `ll_dma_enable_channel()`.
- Parameter setting type: These APIs are used to set feature parameters of peripherals. Example: `ll_dma_set_data_transfer_direction()`.
- Flag and state type: These APIs used to indicate the flag and state of peripheral registers. Example: `ll_i2c_is_active_flag_stop_det()`.
- Interrupt control type: These APIs enable/disable certain peripheral interrupts. Example: `ll_i2c_enable_it_stop_det()`.

- DMA control type: These APIs enable/disable DMA requests from peripherals. Example: `ll_i2c_enable_dma_req_tx()`.

1.3.2 Extension APIs

The extension APIs offer extensional functions that are unavailable on generic APIs for a certain SoC series. The classification rules of extension APIs are consistent with those of generic APIs for both HAL and LL drivers.

1.4 Driver Naming Rules

The driver naming rules of GR551x SoCs comprise general naming rules, HAL API naming rules, and LL API naming rules.

1.4.1 General Naming Rules

The general naming rules apply to HAL and LL drivers, regulating files, modules, structures, and macros.

The rule details are listed in the table below.

Table 1-5 General naming rules

| Category | Name Format | Description | Example |
|------------------|---------------------------------------|---|---|
| File | ccc_ddd_ppp.c/h ccc_ddd_ppp_ex.c/h | Composed of SoC model, driver type, and peripheral name. For extension drivers, add a suffix, <code>_ex</code> , to the end of the name. | gr55xx_hal_gpio.(c/h) gr55xx_hal_gpio_ex.(c/h) gr55xx_ll_gpio.(c/h) |
| Module | HAL_PPP_MODULE | Composed of driver type and peripheral name ending with a suffix of <code>_MODULE</code> . | HAL_I2C_MODULE |
| Macro | PPP_PARAM LL_PPP_PARAM | A macro name should be in upper case. For LL driver macros, put <code>LL_</code> before the peripheral name. | UART_DATABITS_8 LL_UART_PARITY_NONE |
| Structure | ppp_sss_t ll_ppp_sss_t | Composed of peripheral name and structure type with a suffix of <code>_t</code> . For LL driver structures, add <code>ll_</code> to be beginning of the name. | qspi_handle_t ll_uart_init_t |
| Enum | ddd_ppp_enumname_t | Composed of driver type, peripheral name, and enumeration type ending with a suffix of <code>_t</code> . | hal_uart_state_t |
| Enumeration flag | DDD_PPP_ENUM | In upper case, composed of driver type, peripheral name, and flag meaning. | HAL_UART_STATE_RESET |

| Category | Name Format | Description | Example |
|--------------------|--------------|---|-------------|
| Register | REGISTERNAME | A register name, in upper case, should comply with regulations in <i>GR551x Datasheet</i> (excessively long register names are presented in abbreviations). | MODEM_CTRL |
| Register structure | ppp_regs_t | Composed of peripheral name and a suffix of _regs_t. | uart_regs_t |

Naming note:

- ccc: SoC family name. Example: gr55xx
- DDD/ddd: driver type. Example: HAL/hal and LL/ll
- PPP/ppp: peripheral name. Example: GPIO/gpio, QSPI/qspi, and UART/uart
- sss: structure type. Example: handle and init
- PARAM: peripheral parameter
- ENUM: enumeration flag
- REGISTERNAME: register name

1.4.2 Naming Rules of HAL Driver APIs

The table below lists the naming rules of HAL driver APIs.

Table 1-6 Naming rules of HAL driver APIs

| API Type | Name Format | Description |
|--|--------------------------------|--|
| Initialization | hal_ppp_init hal_ppp_deinit | An initialization API function name is composed of HAL driver type and peripheral name ending with _init or _deinit. |
| I/O operation | hal_ppp_operate | Composed of HAL driver type, peripheral name, and operating mode (TX, RX, and callback). |
| | hal_ppp_command_operate | For command-related APIs, add _command between the peripheral name and operating mode. |
| | hal_ppp_operate_it | For I/O operation APIs in interrupt mode, add a suffix of _it. |
| | hal_ppp_operate_dma | For I/O operation APIs in DMA mode, add a suffix of _dma. |
| Interrupt handling and callback function | hal_ppp_irq_handler | Composed of HAL driver type and peripheral name ending with _irq_handler. |

| API Type | Name Format | Description |
|-----------------|--|--|
| | hal_ppp_operate_cplt_callback | For callback API functions that have been completed, add a suffix of _cplt_callback. |
| | hal_ppp_error_callback | Composed of HAL driver type and peripheral name ending with _error_callback. |
| Control | hal_ppp_set_parameter hal_ppp_get_parameter | Composed of HAL driver type, peripheral name, and parameter name. |
| State and error | hal_ppp_get_state hal_ppp_get_error | Composed of HAL driver type, peripheral name, and state or error. |

Note:

- PPP/ppp: peripheral name. Example: QSPI/qspi and UART/uart
- operate: operating mode. Example: transmit/tx, receive/rx, and abort
- parameter: parameter name. Example: fifo_threshold and timeout

The table below is an example showing the HAL API naming rules when QSPI serves as a peripheral.

Table 1-7 Naming rules of HAL driver APIs for QSPI

| API Type | Function Name | Description |
|----------------------|------------------------------|---|
| Initialization | hal_qspi_init | This function initializes QSPI and sets clocks and pin multiplexing. |
| | hal_qspi_deinit | This function deinitializes QSPI and restores it to initial settings. |
| | hal_qspi_mspinit | This function initializes GPIOs, NVIC interrupts, and DMA used by QSPI. |
| | hal_qspi_mspdeinit | This function deinitializes GPIOs, NVIC interrupts, and DMA used by QSPI. |
| I/O operation | hal_qspi_command_transmit | This function enables data transmission and reception in polling mode. |
| | hal_qspi_command_receive | |
| | hal_qspi_command | |
| | hal_qspi_transmit | |
| | hal_qspi_receive | This function enables data transmission and reception in interrupt mode. |
| | hal_qspi_command_transmit_it | |
| | hal_qspi_command_receive_it | |
| | hal_qspi_command_it | |
| hal_qspi_transmit_it | | |

| API Type | Function Name | Description |
|--|--------------------------------|--|
| | hal_qspi_receive_it | This function enables data transmission and reception in DMA mode. |
| | hal_qspi_command_transmit_dma | |
| | hal_qspi_command_receive_dma | |
| | hal_qspi_command_dma | |
| | hal_qspi_transmit_dma | |
| | hal_qspi_receive_dma | |
| | hal_qspi_abort | This function aborts ongoing data transmissions. |
| | hal_qspi_abort_it | |
| Interrupt handling and callback function | hal_qspi_irq_handler | Interrupt handler |
| | hal_qspi_tx_cplt_callback | TX complete callback function |
| | hal_qspi_rx_cplt_callback | RX complete callback function |
| | hal_qspi_error_callback | Error detection callback function |
| | hal_qspi_abort_cplt_callback | Abort complete callback function |
| Control | hal_qspi_set_timeout | This function sets a timeout duration. |
| | hal_qspi_set_tx_fifo_threshold | This function sets a FIFO threshold. |
| | hal_qspi_set_rx_fifo_threshold | |
| | hal_qspi_get_tx_fifo_threshold | This function reads a FIFO threshold. |
| | hal_qspi_get_rx_fifo_threshold | |
| State and error | hal_qspi_get_state | This function reads the peripheral state. |
| | hal_qspi_get_error | This function reads error code. |

1.4.3 Naming Rules of LL Driver APIs

The table below lists the naming rules of LL driver APIs.

Table 1-8 Naming rules of LL driver APIs

| API Type | Name Format | Description |
|--------------------|----------------------------|---|
| Initialization | ll_ppp_init | Composed of LL driver type and peripheral name ending with _init or _deinit. |
| | ll_ppp_deinit | |
| Feature enablement | ll_ppp_enable_function | Composed of LL driver type, peripheral name, state (enable/disable/is_enabled), and function name. |
| | ll_ppp_disable_function | |
| | ll_ppp_is_enabled_function | |
| I/O operation | ll_ppp_transmit_dataN | Composed of LL driver type, peripheral name, data transfer direction (transmit/receive), and data(bit width). |
| | ll_ppp_receive_dataN | |
| Parameter setting | ll_ppp_set_parameter | Composed of LL driver type, peripheral name, parameter operation (set/get), and parameter name. |
| | ll_ppp_get_parameter | |

| API Type | Name Format | Description |
|-------------------|---|---|
| Flag and state | ll_ppp_is_active_flag_flagname ll_ppp_clear_flag_flagname ll_ppp_clear_flag ll_ppp_clear_flagtype_flag ll_ppp_get_flagtype_flag | Two approaches are used to name a flag and state API: <ul style="list-style-type: none"> LL driver type, peripheral name, is_active/clear, _flag, and flag name. The segment of is_active and clear is used to judge the peripheral state or clear the flag. LL driver type, peripheral name, get/clear, flag type, and _flag. The segment of get and clear is used to retrieve or clear a type of flag. Some peripherals do not contain a flag type. |
| Interrupt control | ll_ppp_enable_it_itname ll_ppp_disable_it_itname ll_ppp_is_enabled_it_itname ll_ppp_enable_it ll_ppp_disable_it ll_ppp_is_enabled_it | Composed of LL driver type, peripheral name, state (enable/disable/is_enabled), it, and interrupt name. Functions without an interrupt name (example: ll_ppp_enable_it) can be used to control multiple interrupts. |
| DMA control | ll_ppp_enable_dma_req_tx/rx ll_ppp_disable_dma_req_tx/rx ll_ppp_is_enabled_dma_req_tx/rx | Composed of LL driver type, peripheral name, state (enable/disable/is_enabled), and DMA request type. |

 **Note:**

- PPP/ppp: peripheral name. Example: QSPI/qspi and UART/uart
- function: functionality name. Example: general_call (for I2C)
- N: data bit width, range: 8, 16, and 32
- parameter: parameter name. Example: fifo_threshold and timeout
- flagname: flag name. For example, the flag name of the STOP_DET interrupt in I2C is stop_det.
- flagtype: type of flag that needs to be cleared or got. Example: it and line_status
- itname: interrupt name. For example, the itname of the RDA interrupt in UART is rda.

The table below is an example showing the LL API naming rules when UART serves as a peripheral.

Table 1-9 Naming rules of LL driver APIs for UART

| API Type | Function Name | Description |
|--------------------|-------------------------|--|
| Initialization | ll_uart_init | This function initializes UART and sets parameters such as baud rate and data bit. |
| | ll_uart_deinit | This function deinitializes UART and restores it to initial settings. |
| Feature enablement | ll_uart_enable_fifo | This function controls and judges the FIFO enablement state. |
| | ll_uart_disable_fifo | |
| | ll_uart_is_enabled_fifo | |

| API Type | Function Name | Description |
|-------------------|--------------------------------|---|
| I/O operation | ll_uart_transmit_data8 | This function enables the peripheral to transmit data by byte. |
| | ll_uart_receive_data8 | This function enables the peripheral to receive data by byte. |
| Parameter setting | ll_uart_set_parity | This function sets the odd-even parity bit. |
| | ll_uart_get_parity | This function gets the odd-even parity bit. |
| Flag and state | ll_uart_is_active_flag_rff | This function is used to judge whether the RFF flag is set. |
| | ll_uart_get_line_status_flag | This function is used to get the line state. |
| | ll_uart_clear_line_status_flag | This function is used to clear the line state. |
| | ll_uart_get_it_flag | This function is used to get the interrupt state. |
| Interrupt control | ll_uart_enable_it_rda | This function controls and judges the state of the received data available (RDA) interrupt. |
| | ll_uart_disable_it_rda | |
| | ll_uart_is_enabled_it_rda | |
| | ll_uart_enable_it | This function controls and judges the states of multiple interrupts. |
| | ll_uart_disable_it | |
| | ll_uart_is_enabled_it | |
| DMA control | None | The DMA requests of UART are managed by hardware. No settings are required on software. |

1.5 Data Structure

Each HAL driver contains the following data structures:

- Peripheral handle structure
- Initialization structure
- Configuration structure

To simplify parameter setting in scenarios where only LL drivers are used, Goodix defines initialization structures in LL drivers.

1.5.1 Peripheral Handle Structure

The HAL drivers adopt a multi-instance architecture that allows working with several instances simultaneously on a peripheral. `ppp_handle_t *handle` is the main structure in the architecture. It defines the handle of each instance and stores the peripheral setting parameters, register structure pointers, and diverse run-time variables of each instance.

The peripheral handle is used for the purposes below:

- Multi-instance support: Each peripheral instance has its own handle, which results in independent peripheral setting parameters and run-time variables for each instance.
- Intra-API communications: The handle stores shared variables during peripheral operation, which enables data exchange between APIs.

- **Storage:** The handle stores and manages global variables of a specific peripheral driver.

An example of Serial Peripheral Interface (SPI) peripheral structure is shown below:

```
typedef struct
{
    ssi_regs_t    *p_instance; /**< SPI registers base address*/
    spi_init_t    init;        /**< SPI communication parameters*/
    uint8_t       *p_tx_buffer; /**< Pointer to SPI Tx transfer Buffer */
    __IO uint32_t tx_xfer_size; /**< SPI Tx Transfer size*/
    __IO uint32_t tx_xfer_count; /**< SPI Tx Transfer Counter*/
    uint8_t       *p_rx_buffer; /**< Pointer to SPI Rx transfer Buffer */
    __IO uint32_t rx_xfer_size; /**< SPI Rx Transfer size*/
    __IO uint32_t rx_xfer_count ; /**< SPI Rx Transfer Counter*/
    void (*write_fifo)(struct _spi_handle *p_spi); /**< Pointer to SPI Tx transfer
                                                    FIFO write function */
    void (*read_fifo)(struct _spi_handle *p_spi); /**< Pointer to SPI Rx transfer
                                                    FIFO read function */

    dma_handle_t  *p_dmatx; /**< SPI Tx DMA Handle parameters*/
    dma_handle_t  *p_dmarx; /**< SPI Rx DMA Handle parameters*/
    __IO hal_lock_t lock; /**< Locking object*/
    __IO hal_spi_state_t state; /**< SPI communication state*/
    __IO uint32_t error_code; /**< SPI Error code*/
    uint32_t      timeout; /**< timeout for the SPI memory access*/
} spi_handle_t;
```

Note:

No handle structure is used for system peripherals that are shared by multiple modules. Examples include GPIO, System Tick Timer (SysTick), NVIC, and PWR.

1.5.2 Initialization Structure

The initialization structure is used to store setting parameters in initializing peripherals.

An example of UART initialization structure is shown below. The structure can set baud rate, data bit, stop bit, hardware flow control mode, and access (RX) timeout.

```
typedef struct
{
    uint32_t baud_rate;
    uint32_t data_bits;
    uint32_t stop_bits;
    uint32_t parity;
    uint32_t hw_flow_ctrl;
    uint32_t rx_timeout_mode;
} uart_init_t;
```

1.5.3 Configuration Structure

The configuration structure is used to configure parameters of sub-modules and sub-instances.

An example of channel initialization structure for PWM is shown below:

```
typedef struct
{
    uint8_t duty; /**< Specifies the duty in PWM output mode. This parameter must be a number
                  between 0 ~ 100.*/
```

```
uint8_t drive_polarity; /**< Specifies the drive polarity in PWM output mode.This parameter  
can be a value of @ref PWM_DRIVEPOLARITY.*/  
} pwm_channel_init_t;
```

2 HAL Drivers

2.1 Introduction

This section introduces common HAL driver resources of peripherals and methods on how to use HAL drivers.

2.1.1 HAL Common Resources

In HAL drivers of GR551x SoCs, the common resources of all peripherals including common enumerations, structures, and macros are defined in *gr55xx_hal_def.h*. The details are as follows:

- HAL status: The HAL status indicates the operating state of all APIs except for Boolean functions and interrupt handlers. It has four values as below:

```
typedef enum
{
    HAL_OK          = 0x00U,
    HAL_ERROR       = 0x01U,
    HAL_BUSY        = 0x02U,
    HAL_TIMEOUT     = 0x03
} hal_status_t;
```

- HAL lock: The HAL lock prevents unauthorized access to shared resources. It has two values as below:

```
typedef enum
{
    HAL_UNLOCKED = 0x00U,
    HAL_LOCKED   = 0x01
} hal_lock_t;
```

- Common macros: Common macros comprise maximum delay macro (HAL_MAX_DELAY), the macro linking a peripheral to a DMA instance handle (HAL_LINKDMA), and a macro to delete alarms indicating that some parameters are not used in compiling. An example is defined below:

```
#define HAL_MAX_DELAY          0xFFFFFFFFU

#define __HAL_LINKDMA(__HANDLE__, __PPP_DMA_FIELD__, __DMA_HANDLE__) \
do{ \
    (__HANDLE__)->__PPP_DMA_FIELD__ = &(__DMA_HANDLE__); \
    (__DMA_HANDLE__).p_parent = (__HANDLE__); \
} while(0U)

#define UNUSED(x) ((void)(x))
```

2.1.2 How to Use HAL Drivers

The figure below shows the calling process of HAL drivers.

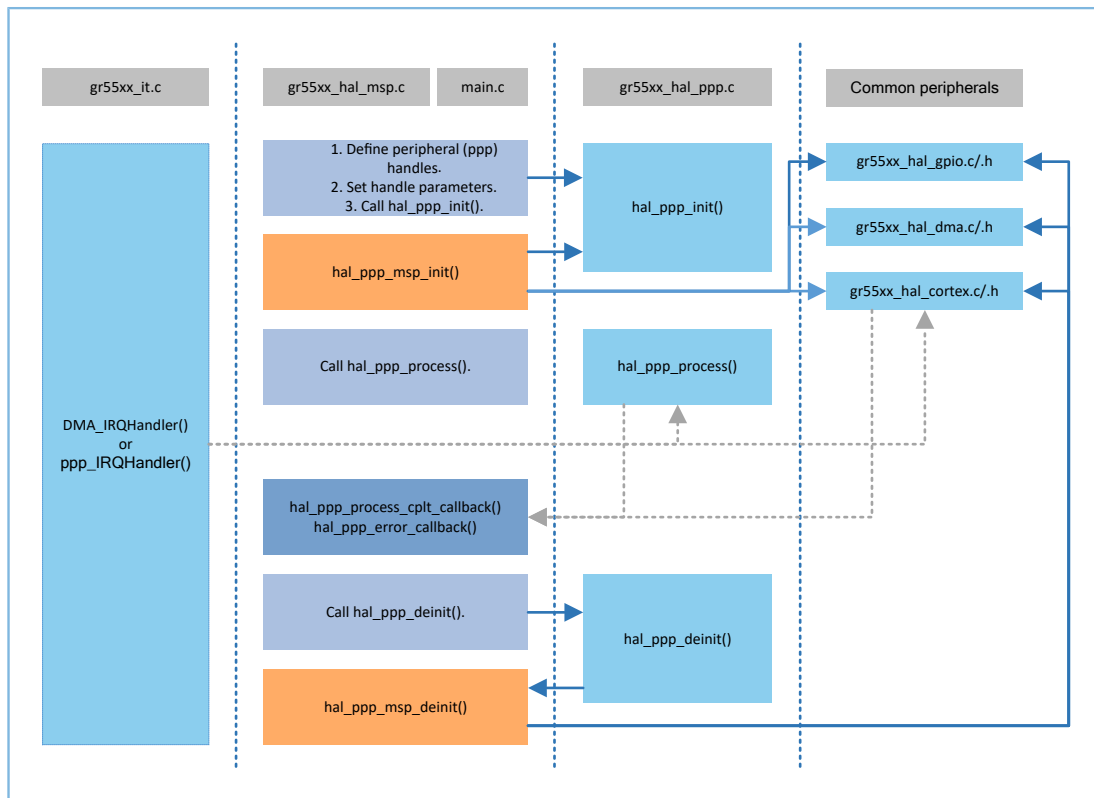


Figure 2-1 Calling process of HAL drivers

Note:

- indicates functions that are implemented in HAL drivers.
- indicates code that should be implemented in user applications by developers.
- indicates msp (MCU Specific Package) functions that should be implemented in user applications.
- indicates functions called in interrupt handlers.

Detailed process description:

- Developers overwrite msp functions: `hal_ppp_msp_init()` and `hal_ppp_msp_deinit()` of peripherals (ppp) in user application files (such as `main.c` and `gr55xx_msp.c`).
- If interrupt APIs are required, developers need to overwrite the corresponding callback functions such as `hal_ppp_process_cplt_callback()`.
- Developers define handles of peripherals and configure related parameters in user application files.
- Developers initialize peripherals (ppp) by calling `hal_ppp_init()` implemented in ppp driver files. During initialization, the overwritten `hal_ppp_msp_init()` function is called to initialize GPIO pins, NVIC interrupts, and DMA channels used by ppp peripherals.
- The HAL driver calling process varies depending on I/O operation modes: polling, interrupt, and DMA.

- In polling mode, developers call `hal_ppp_process()` to perform I/O operations. The function is considered successful after the I/O operations complete.
 - In interrupt mode, developers call `hal_ppp_process_it()` to perform I/O operations. The function is considered successful after interrupt is enabled. The I/O operations are implemented in `PPP_IRQHandler()`, after which overwritten callback functions are called to push notifications indicating the event has been completed.
 - In DMA mode, developers call `hal_ppp_process_dma()` to perform I/O operations. The function is considered successful when data transmission begins. The I/O operations are implemented in DMA peripherals, after which `DMA_IRQHandler()` calls the overwritten callback functions to push notifications indicating the event has been completed.
6. After peripherals are used, developers can call `hal_ppp_deinit()` in ppp driver files to deinitialize the ppp and restore the corresponding registers to default values. During deinitialization, the overwritten `hal_ppp_msp_deinit()` function is called to deinitialize GPIO pins, NVIC interrupts, and DMA channels used by peripherals (ppp).

The sections below elaborate on the initialization, I/O operation, timeout detection, and error check mechanisms of HAL drivers.

2.1.2.1 HAL Driver Initialization

2.1.2.1.1 Global Initialization

The `gr55xx_hal.c` file provides a set of APIs to initialize/deinitialize the SysTick in HAL drivers, allowing peripheral drivers to detect timeout in data transmission and reception in polling mode.

- `hal_init()`: This function can be called after SoC startup to perform the following:
 - `hal_msp_init()`: Call this function to initialize clock, GPIO pins, interrupts, and DMA.
- `hal_deinit()`: Call `hal_msp_deinit()` to deinitialize clock, GPIO pins, interrupts, and DMA.

2.1.2.1.2 MSP Initialization

During peripheral initialization, the `hal_ppp_init()` calls the `hal_ppp_msp_init()` function to initialize the GPIO pins, interrupts, and DMA used by the peripheral. During peripheral deinitialization, the `hal_ppp_msp_deinit()` function is called. Both `hal_ppp_msp_init()` and `hal_ppp_msp_deinit()` are declared empty as weak functions in HAL drivers. Therefore, developers need to overwrite the two functions on demand in actual use. The function details are described as below:

```
__weak void hal_ppp_msp_init(ppp_handle_t *p_ppp)
{
    /* Prevent unused argument(s) compilation warning */
    UNUSED(p_ppp);
}

__weak void hal_ppp_msp_deinit(ppp_handle_t *p_ppp)
{
    /* Prevent unused argument(s) compilation warning */
```

```
UNUSED(p_ppp);  
}
```

2.1.2.2 HAL Driver I/O Operations

The HAL drivers provide three I/O operation (data read/write) modes for peripherals: polling, interrupt, and DMA.

2.1.2.2.1 Polling Mode

In polling mode, data reads and writes are processed in a loop, which means the read/write APIs return a process status after the data read/write completes. A data read/write is considered successful when the read/write API returns the HAL_OK status. Otherwise, it returns HAL_ERROR or HAL_TIMEOUT. Users can retrieve the specific status and error code through hal_ppp_get_state() and hal_ppp_get_error(). To prevent an infinite loop in read/write API processing, the HAL drivers adopt a timeout detection mechanism. Users can specify a timeout period in timeout parameters in APIs.

The example below shows a typical read/write API processing sequence in polling mode:

```
hal_status_t hal_ppp_transmit(ppp_handle_t *p_ppp, uint8_t *p_data, uint16_t size, uint32_t  
                             timeout)  
{  
    if ((NULL == p_data) || (0U == size))  
    {  
        return HAL_ERROR;  
    }  
    (...)  
    while (data processing is running)  
    {  
        if (timeout reached)  
        {  
            return HAL_TIMEOUT;  
        }  
    }  
    (...)  
    return HAL_OK;  
}
```

2.1.2.2.2 Interrupt Mode

In interrupt mode, data reads and writes are not processed in a loop. This means the read/write APIs return a process status when the read/write interrupt is enabled. The read/write operations are implemented in peripheral interrupt handlers. The read/write operation is considered successful when the application calls the callback function customized by developers to notify the application of the process status. Developers can retrieve the current read/write status through the hal_ppp_get_state() function.

In interrupt mode, the HAL drivers define the following API functions:

- hal_ppp_process_it(): read/write API function in interrupt mode
- hal_ppp_irq_handler(): peripheral interrupt handler
- _weak hal_ppp_process_cplt_callback(): process complete callback function; implemented by developers
- _weak hal_ppp_process_error_callback(): process error callback function; implemented by developers

To use APIs in interrupt mode, developers need to register the interrupt handler, `hal_ppp_irq_handler()`, in the `gr55xx_it.c` file, after which the `hal_ppp_process_it()` can be called to perform data reads or writes.

Callback functions are declared as weak functions in HAL drivers, which means developers need to customize callback functions to release cache memories after data reads/writes complete.

The example below shows the read/write API used by UART0 in interrupt mode:

In the `main.c` file:

```
uart_handle_t uart_handle;
int main(void)
{
    uart_handle.p_instance      = UART0;
    uart_handle.init.baud_rate  = 115200;
    uart_handle.init.data_bits  = UART_DATABITS_8;
    uart_handle.init.stop_bits  = UART_STOPBITS_1;
    uart_handle.init.parity     = UART_PARITY_NONE;
    uart_handle.init.hw_flow_ctrl = UART_HWCONTROL_NONE;
    hal_uart_init(&uart_handle);

    char *p_tx_buff = "Hello World!\r\n";
    hal_uart_transmit_it(&uart_handle, p_tx_buff, strlen((char*)p_tx_buff));
    while (hal_uart_get_state(&uart_handle) != HAL_UART_STATE_READY);

    hal_uart_deinit(&uart_handle);
}

void hal_uart_tx_cplt_callback(uart_handle_t *p_uart)
{
    (...)
}

void hal_uart_error_callback(uart_handle_t *p_uart)
{
    (...)
}
```

In the `gr55xx_it.c` file:

```
void UART0_IRQHandler(void)
{
    Hal_uart_irq_handler(&uart_handle);
}
```

Note:

`UART0_IRQHandler()` can be placed directly in the `main.c` file.

2.1.2.2.3 DMA Mode

In DMA mode, data reads and writes are not processed in a loop. The read/write operation is considered successful when the interrupt handler calls the corresponding callback function to notify the application of the process status. Developers can retrieve the current read/write status through the `hal_ppp_get_state()` function.

In DMA mode, the HAL drivers define the following API functions:

- `hal_ppp_process_dma()`: read/write API function in DMA mode

- `hal_ppp_irq_handler()`: peripheral interrupt handler
- `_weak hal_ppp_process_cplt_callback()`: process complete callback function; implemented by developers
- `_weak hal_ppp_process_error_callback()`: process error callback function; implemented by developers

To use APIs in DMA mode, developers need to (1) register the interrupt handler, `hal_dma_irq_handler()`. For some peripherals such as I2C, registration of `hal_ppp_irq_handler()` is required; (2) initialize the required DMA channels for initialization in the `hal_ppp_msp_init()` function; (3) call the `hal_ppp_process_dma()` function to process data reads and writes.

The example below shows the read/write API used by UART0 in DMA mode:

- In the *main.c* file:

```
uart_handle_t uart_handle;
int main(void)
{
    uart_handle.p_instance          = UART0;
    uart_handle.init.baud_rate      = 115200;
    uart_handle.init.data_bits      = UART_DATABITS_8;
    uart_handle.init.stop_bits      = UART_STOPBITS_1;
    uart_handle.init.parity         = UART_PARITY_NONE;
    uart_handle.init.hw_flow_ctrl   = UART_HWCONTROL_NONE;
    hal_uart_init(&uart_handle);

    char *p_tx_buff = "Hello World!\r\n";
    hal_uart_transmit_dma(&uart_handle, p_tx_buff, strlen((char*)p_tx_buff));
    while (hal_uart_get_state(&uart_handle) != HAL_UART_STATE_READY);
    hal_uart_deinit(&uart_handle);
}

void hal_uart_tx_cplt_callback(uart_handle_t *p_uart)
{
    (...)
}

void hal_uart_error_callback(uart_handle_t *p_uart)
{
    (...)
}
```

- In the *gr55xx_hal_msp.c* file:

```
void hal_uart_msp_init (uart_handle_t *p_uart)
{
    static dma_handle_t hdma_tx;
    static dma_handle_t hdma_rx;
    (...)
    hal_dma_init(&hdma_tx);
    hal_dma_init(&hdma_rx);

    __HAL_LINKDMA(p_uart, p_dmatx, hdma_tx);
    __HAL_LINKDMA(p_uart, p_dmarx, hdma_rx);
    (...)
}
```

- In the *gr55xx_it.c* file:


```

void UART0_IRQHandler(void)
{
    hal_uart_irq_handler(&uart_handle);
}
void DMA_IRQHandler(void)
{
    Hal_uart_irq_handler(uart_handle.p_dmatx);
}

```

Note:

hal_uart_msp_init() and UART0_IRQHandler() can be placed directly in the *main.c* file.

2.1.2.3 Timeout Detection and Error Check

2.1.2.3.1 Timeout Detection

To properly use APIs in polling mode, the HAL drivers provide a timeout detection mechanism to prevent the SoC from entering an infinite loop due to errors. The example below shows the data transmission API used by SPI in polling mode:

```

hal_status_t hal_spi_transmit(spi_handle_t *p_spi, uint8_t *p_data, uint32_t length,
                             uint32_t timeout)

```

The timeout parameter value represents the maximum timeout period allowed for data transmission. When the actual timeout exceeds the preset timeout value, the API function returns the HAL_TIMEOUT status.

The HAL driver timeout value is defined in the *gr55xx_hal_def.h* file, ranging from zero to HAL_MAX_DELAY (0xFFFFFFFF). The detailed timeout values are listed in [Table 2-1](#).

Table 2-1 Timeout values

| Timeout Value | Description |
|--------------------------|---|
| 0 | No timeout or waiting. Immediate loop exit if flag detection conditions are not met |
| 1 to (HAL_MAX_DELAY - 1) | Timeout period (ms) |
| HAL_MAX_DELAY | Infinite loop until the process is successful |

In addition, some peripherals use fixed timeouts in certain cases. For example, I2C is detected as busy when the timeout reaches 25 milliseconds. In comparison to timeouts input in API functions, fixed timeouts are implemented by using internal macros in APIs. These timeouts cannot be modified.

2.1.2.3.2 Error Check

To improve the robustness of HAL drivers and prevent unexpected errors, the HAL drivers provide an error check mechanism.

- Validity check on input parameters

Ensure the input parameters provided for developers in API functions are pre-defined and valid. Otherwise, the application may break down or enter an undefined state. Therefore, the HAL drivers of GR551x SoCs introduce validity check on input parameters.

For example, the HAL drivers check the validity of cache pointer and length of the parameter input in the `hal_uart_transmit_it()` function. The detailed code is shown as below:

```
hal_status_t hal_uart_transmit_it(uart_handle_t *p_uart, uint8_t *p_data, uint16_t size)
{
    /* Check that a Tx process is not already ongoing */
    if(HAL_UART_STATE_READY == p_uart->g_state)
    {
        if((NULL == p_data ) || (0U == size))
        {
            return HAL_ERROR;
        }
        (...)
    }
}
```

- Handle validity check

Peripheral handles are the most important part in HAL drivers because they store configuration parameters and run-time variables of peripheral drivers. The HAL drivers of GR551x SoCs implement validity check on peripheral handles in the `hal_ppp_init()` function.

For example, the HAL drivers check the validity of UART handles in the `hal_uart_init` function. The detailed code is shown as below:

```
hal_status_t hal_uart_init(uart_handle_t *p_uart)
{
    /* Check the UART handle allocation */
    if(NULL == p_uart)
    {
        return HAL_ERROR;
    }
    (...)
}
```

- Timeout error check

The HAL drivers check the run-time of API functions in polling mode. A timeout status is returned when the operation lasts a period that is longer than the preset value.

The example below shows how the HAL drivers check the timeout of UART in reception data in polling mode.

```
hal_status_t hal_uart_receive(uart_handle_t *p_uart, uint8_t *p_data, uint16_t size,
                             uint32_t timeout)
{
    (...)
    /* as long as data have to be received */
    while(0U < p_uart->rx_xfer_count)
    {
        if(HAL_OK != uart_wait_fifo_flag_until_timeout(p_uart, UART_FLAG_FIFO_RFNE, RESET,
                                                       tickstart, timeout)
        {

```

```

        return HAL_TIMEOUT;
    }
    (...)
}
(...)
}

```

In all peripheral handles, a global variable, `error_code`, is defined to store the error code in API operations. This allows developers to call `hal_ppp_get_error()` to retrieve the specific error type when the API function returns the `HAL_ERROR` status. An example is displayed as below:

```

uint32_t hal_uart_get_error(uart_handle_t *p_uart)
{
    return p_uart -> error_code;
}

```

2.1.2.3.3 Run-time Parameter Check

The HAL drivers provide run-time check for input parameters, checking whether an input parameter is within the allowed range. The run-time check is implemented by the `gr_assert_param` macro which is defined in the `gr55xx_hal_conf.h` file. Developers can use `USE_FULL_ASSERT` macro to enable or disable the run-time check. The example below shows I2C instance, speed, local device address, and addressing mode are checked in the initialization function of I2C.

```

hal_status_t hal_i2c_init(i2c_handle_t *p_i2c)
{
    (...)
    /* Check the parameters */
    gr_assert_param(IS_I2C_ALL_INSTANCE(p_i2c->instance));
    gr_assert_param(IS_I2C_SPEED(p_i2c->init.speed));
    gr_assert_param(IS_I2C_OWN_ADDRESS(p_i2c->init.own_address));
    gr_assert_param(IS_I2C_ADDRESSING_MODE(p_i2c->init.addressing_mode));
    gr_assert_param(IS_I2C_GENERAL_CALL(p_i2c->init.general_call_mode));
    (...)
}

```

If the expression passed to the `gr_assert_param` macro is false, the `assert_failed()` function is called and returns the name of the false file and line number of the call that failed. The `assert_failed()` function is customized by developers. The macro statements of the `gr_assert_param` macro are displayed as below:

```

#ifdef USE_FULL_ASSERT
/**
 * @brief The gr_assert_param macro is used for function's parameters check.
 * @param expr If expr is false, it calls assert_failed function
 *            which reports the name of the source file and the source
 *            line number of the call that failed.
 *            If expr is true, it returns no value.
 * @retval None
 */
#define gr_assert_param(expr) ((expr) ? (void)0U : assert_failed((char *)__FILE__, \
    __LINE__))
/* Exported functions ----- */
void assert_failed(char* file, uint32_t line);
#else
#define gr_assert_param(expr) ((void)0U)

```

```
#endif /* USE_FULL_ASSERT */
```

2.2 HAL Cortex Generic Driver

2.2.1 Cortex Driver Functionalities

Based on secondary encapsulation of NVIC-related APIs in the *core_cm4.h* file, the HAL Cortex driver offers a set of interrupt control and SysTick configuration API functions with the following functionalities:

- Configure and manage interrupt priorities.
- Enable/Disable interrupts.
- Pend, clear, and query interrupts.
- Initialize the SysTick and set the source clock.

2.2.2 How to Use Cortex Driver

Developers can:

1. Configure interrupt priority groups by calling `hal_nvic_set_priority_grouping()` in the overwritten `hal_msp_init()` function.
2. Configure peripheral interrupt priorities and enable peripheral interrupts by calling `hal_nvic_clear_pending_irq()`, `hal_nvic_enable_irq()`, and `hal_nvic_set_priority()` in the overwritten `hal_ppp_msp_init()` function.
3. Disable peripheral interrupts by calling `hal_nvic_disable_irq()` in the overwritten `hal_ppp_msp_deinit()` function.
4. Initialize the SysTick by calling `hal_systick_config()`.

2.2.3 Cortex Driver APIs

The Cortex driver APIs are listed in the table below:

Table 2-2 HAL Cortex driver APIs

| API Type | API Name | Description |
|----------------|---|--|
| Initialization | <code>hal_nvic_set_priority_grouping()</code> | Set the interrupt priority grouping field. |
| | <code>hal_nvic_set_priority()</code> | Set the priority of an interrupt. |
| | <code>hal_nvic_enable_irq()</code> | Enable an interrupt. |
| | <code>hal_nvic_disable_irq()</code> | Disable an interrupt. |
| | <code>hal_nvic_system_reset()</code> | Reset the SoC. |
| | <code>hal_systick_config()</code> | Configure the SysTick. |
| Control | <code>hal_nvic_get_priority_grouping()</code> | Get the interrupt priority grouping field. |
| | <code>hal_nvic_get_priority()</code> | Get the priority of an interrupt. |
| | <code>hal_nvic_set_pending_irq()</code> | Set the pending bit of an interrupt. |
| | <code>hal_nvic_get_pending_irq()</code> | Get the pending bit of an interrupt. |

| API Type | API Name | Description |
|---------------------------------|---------------------------------|--|
| | hal_nvic_clear_pending_irq() | Clear the pending bit of an interrupt. |
| | hal_nvic_get_active() | Get an active interrupt. |
| | hal_systick_clk_source_config() | Set the SysTick clock source. |
| Interrupt handling and callback | hal_systick_irq_handler() | Interrupt handler |
| | hal_systick_callback() | Interrupt callback |

The sections below elaborate on these APIs.

2.2.3.1 hal_nvic_set_priority_grouping

Table 2-3 hal_nvic_set_priority_grouping API

| | |
|-----------------------------|--|
| Function Prototype | void hal_nvic_set_priority_grouping(uint32_t priority_group) |
| Function Description | Set the interrupt priority grouping field. |
| Parameter | <p>priority_group: the interrupt priority grouping field. This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • NVIC_PRIORITYGROUP_0 (0 bit for preemption priority, 8 bits for subpriority) • NVIC_PRIORITYGROUP_1 (1 bit for preemption priority, 7 bits for subpriority) • NVIC_PRIORITYGROUP_2 (2 bits for preemption priority, 6 bits for subpriority) • NVIC_PRIORITYGROUP_3 (3 bits for preemption priority, 5 bits for subpriority) • NVIC_PRIORITYGROUP_4 (4 bits for preemption priority, 4 bits for subpriority) • NVIC_PRIORITYGROUP_5 (5 bits for preemption priority, 3 bits for subpriority) • NVIC_PRIORITYGROUP_6 (6 bits for preemption priority, 2 bits for subpriority) • NVIC_PRIORITYGROUP_7 (7 bits for preemption priority, 1 bit for subpriority) |
| Return Value | None |
| Remarks | When the priority_group is set to NVIC_PRIORITYGROUP_0, the preemption priority is unavailable. |

2.2.3.2 hal_nvic_set_priority

Table 2-4 hal_nvic_set_priority API

| | |
|-----------------------------|--|
| Function Prototype | void hal_nvic_set_priority(IRQn_Type IRQn, uint32_t preempt_priority, uint32_t sub_priority) |
| Function Description | Set the preemption priority and subpriority of a specified interrupt request number. |
| Parameter | <p>IRQn: the request number of the interrupt to be set. For details, see the interrupt number tables in the <i>gr551xx.h</i> file.</p> <p>preempt_priority: the preemption priority for the IRQn channel; value range: 0 to 127. For details, see Cortex_NVIC_Priority_Table in the <i>gr551xx.h</i> file.</p> |

| | |
|---------------------|--|
| | sub_priority: the subpriority for the IRQn channel; value range: 0 to 255. For details, see Cortex_NVIC_Priority_Table in the <i>gr55xx_hal_cortex.c</i> file. |
| Return Value | None |
| Remarks | |

2.2.3.3 hal_nvic_enable_irq

Table 2-5 hal_nvic_enable_irq API

| | |
|-----------------------------|---|
| Function Prototype | void hal_nvic_enable_irq(IRQn_Type IRQn) |
| Function Description | Enable the interrupt corresponding to a specified IRQn. |
| Parameter | IRQn: the request number of the interrupt to be enabled. For details, see the interrupt number tables in the <i>gr551xx.h</i> file. |
| Return Value | None |
| Remarks | |

2.2.3.4 hal_nvic_disable_irq

Table 2-6 hal_nvic_disable_irq API

| | |
|-----------------------------|--|
| Function Prototype | void hal_nvic_disable_irq(IRQn_Type IRQn) |
| Function Description | Disable the interrupt corresponding to a specified IRQn. |
| Parameter | IRQn: the request number of the interrupt to be disabled. For details, see the interrupt number tables in the <i>gr551xx.h</i> file. |
| Return Value | None |
| Remarks | |

2.2.3.5 hal_nvic_system_reset

Table 2-7 hal_nvic_system_reset API

| | |
|-----------------------------|----------------------------------|
| Function Prototype | void hal_nvic_system_reset(void) |
| Function Description | Reset the SoC. |
| Parameter | None |
| Return Value | None |
| Remarks | |

2.2.3.6 hal_systick_config

Table 2-8 hal_systick_config API

| | |
|-----------------------------|---|
| Function Prototype | uint32_t hal_systick_config(uint32_t ticks) |
| Function Description | Initialize the SysTick with interrupt enabled, and start the SysTick. |
| Parameter | ticks: the initial tick of the timer; value range: 0x0000_0000 to 0xFFFF_FFFF |
| Return Value | The initialization status. This parameter can be one of the following values: <ul style="list-style-type: none"> • 0: Initialization succeeds. • 1: Initialization fails. |
| Remarks | |

2.2.3.7 hal_nvic_get_priority_grouping

Table 2-9 hal_nvic_get_priority_grouping API

| | |
|-----------------------------|---|
| Function Prototype | uint32_t hal_nvic_get_priority_grouping(void) |
| Function Description | Get the interrupt priority grouping field. |
| Parameter | None |
| Return Value | Interrupt priority grouping field. This parameter can be one of the following values: <ul style="list-style-type: none"> • NVIC_PRIORITYGROUP_0 (0 bit for preemption priority, 8 bits for subpriority) • NVIC_PRIORITYGROUP_1 (1 bit for preemption priority, 7 bits for subpriority) • NVIC_PRIORITYGROUP_2 (2 bits for preemption priority, 6 bits for subpriority) • NVIC_PRIORITYGROUP_3 (3 bits for preemption priority, 5 bits for subpriority) • NVIC_PRIORITYGROUP_4 (4 bits for preemption priority, 4 bits for subpriority) • NVIC_PRIORITYGROUP_5 (5 bits for preemption priority, 3 bits for subpriority) • NVIC_PRIORITYGROUP_6 (6 bits for preemption priority, 2 bits for subpriority) • NVIC_PRIORITYGROUP_7 (7 bits for preemption priority, 1 bit for subpriority) |
| Remarks | |

2.2.3.8 hal_nvic_get_priority

Table 2-10 hal_nvic_get_priority API

| | |
|-----------------------------|---|
| Function Prototype | void hal_nvic_get_priority(IRQn_Type IRQn, uint32_t priority_group, uint32_t *p_preempt_priority, uint32_t *p_sub_priority) |
| Function Description | Get the preemption priority and subpriority of an interrupt corresponding to a specified IRQn based on the priority grouping field. |
| Parameter | IRQn: the request number of the interrupt to be enabled. For details, see the interrupt number tables in the <i>gr551xx.h</i> file. |

| | |
|---------------------|--|
| | <p>priority_group: the interrupt priority grouping field. This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • NVIC_PRIORITYGROUP_0 (0 bit for preemption priority, 8 bits for subpriority) • NVIC_PRIORITYGROUP_1 (1 bit for preemption priority, 7 bits for subpriority) • NVIC_PRIORITYGROUP_2 (2 bits for preemption priority, 6 bits for subpriority) • NVIC_PRIORITYGROUP_3 (3 bits for preemption priority, 5 bits for subpriority) • NVIC_PRIORITYGROUP_4 (4 bits for preemption priority, 4 bits for subpriority) • NVIC_PRIORITYGROUP_5 (5 bits for preemption priority, 3 bits for subpriority) • NVIC_PRIORITYGROUP_6 (6 bits for preemption priority, 2 bits for subpriority) • NVIC_PRIORITYGROUP_7 (7 bits for preemption priority, 1 bit for subpriority) <p>p_preempt_priority: pointer to the unsigned variable type of integer. The variable is used to store the got preemption priority.</p> <p>p_sub_priority: pointer to the unsigned variable type of integer. The variable is used to store the got subpriority.</p> |
| Return Value | None |
| Remarks | |

2.2.3.9 hal_nvic_set_pending_irq

Table 2-11 hal_nvic_set_pending_irq API

| | |
|-----------------------------|--|
| Function Prototype | void hal_nvic_set_pending_irq(IRQn_Type IRQn) |
| Function Description | Set the pending bit of an interrupt corresponding to a specified IRQn. |
| Parameter | IRQn: the request number of the interrupt to be pended. For details, see the interrupt number tables in the <i>gr551xx.h</i> file. |
| Return Value | None |
| Remarks | |

2.2.3.10 hal_nvic_get_pending_irq

Table 2-12 hal_nvic_get_pending_irq API

| | |
|-----------------------------|--|
| Function Prototype | uint32_t hal_nvic_get_pending_irq(IRQn_Type IRQn) |
| Function Description | Get the pending bit of an interrupt corresponding to a specified IRQn. |
| Parameter | IRQn: the request number of the interrupt to be read. For details, see the interrupt number tables in the <i>gr551xx.h</i> file. |
| Return Value | <p>Pending status. This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • 0: Interrupt is not pending. |

| | |
|---------|--|
| | <ul style="list-style-type: none"> 1: Interrupt is pending. |
| Remarks | |

2.2.3.11 hal_nvic_clear_pending_irq

Table 2-13 hal_nvic_clear_pending_irq API

| | |
|----------------------|---|
| Function Prototype | void hal_nvic_clear_pending_irq(IRQn_Type IRQn) |
| Function Description | Clear the pending bit of an interrupt corresponding to a specified IRQn. |
| Parameter | IRQn: the request number of the pending interrupt to be cleared. For details, see the interrupt number tables in the <i>gr551xx.h</i> file. |
| Return Value | None |
| Remarks | |

2.2.3.12 hal_nvic_get_active

Table 2-14 hal_nvic_get_active API

| | |
|----------------------|--|
| Function Prototype | uint32_t hal_nvic_get_active(IRQn_Type IRQn) |
| Function Description | Get an active interrupt corresponding to a specified IRQn. |
| Parameter | IRQn: the request number of the interrupt to be read. For details, see the interrupt number tables in the <i>gr551xx.h</i> file. |
| Return Value | Active status. This parameter can be one of the following values: <ul style="list-style-type: none"> 0: Interrupt has not been processed. 1: Interrupt is being processed. |
| Remarks | |

2.2.3.13 hal_systick_clk_source_config

Table 2-15 hal_systick_clk_source_config API

| | |
|----------------------|--|
| Function Prototype | void hal_systick_clk_source_config(uint32_t clk_source) |
| Function Description | Set the SysTick clock source. |
| Parameter | clk_source: specified clock source. This parameter can be one of the following values: <ul style="list-style-type: none"> SYSTICK_CLKSOURCE_REFCLK (external reference clock) SYSTICK_CLKSOURCE_HCLK (AHB clock) |
| Return Value | None |
| Remarks | |

2.2.3.14 hal_systick_irq_handler

Table 2-16 hal_systick_irq_handler API

| | |
|-----------------------------|------------------------------------|
| Function Prototype | void hal_systick_irq_handler(void) |
| Function Description | Handle SysTick interrupt requests. |
| Parameter | None |
| Return Value | None |
| Remarks | |

2.2.3.15 hal_systick_callback

Table 2-17 hal_systick_callback API

| | |
|-----------------------------|--|
| Function Prototype | void hal_systick_callback(void) |
| Function Description | SysTick interrupt callback |
| Parameter | None |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite this function before using it. |

2.3 HAL System Driver

2.3.1 System Driver Functionalities

The HAL System driver features the following functionalities:

- Configure and enable the SysTick.
- Get the version information of the driver.

Most of the system APIs are declared as weak function, so overwriting on these APIs by developers is required based on actual needs.

2.3.2 How to Use System Driver

Developers can use the system driver in the following scenarios:

1. Call the hal_init() function during application startup to initialize the status of the SysTick.
2. Call the hal_get_hal_version() function to get the version information of the driver.

2.3.3 System Driver APIs

The system driver APIs are listed in the table below:

Table 2-18 System driver APIs

| API Type | API Name | Description |
|-----------------|-----------------------|--|
| Initialization | hal_init() | Initialize the system driver. |
| | hal_deinit() | Deinitialize the system driver. |
| | hal_msp_init() | Initialize the clocks, GPIOs, and interrupts related to the system driver. |
| | hal_msp_deinit() | Deinitialize the clocks, GPIOs, and interrupts related to the system driver. |
| | hal_init_tick() | Initialize the SysTick. |
| Control | hal_suspend_tick() | Suspend the tick increment. |
| | hal_resume_tick() | Resume the tick increment. |
| State and error | hal_get_hal_version() | Get the current HAL driver version. |

The sections below elaborate on these APIs.

2.3.3.1 hal_init

Table 2-19 hal_init API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_init(void) |
| Function Description | Initialize the System driver, and call the hal_init_tick() function to initialize the tick. |
| Parameter | None |
| Return Value | <p>HAL status. This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • HAL_OK (normal) • HAL_ERROR (operation error) • HAL_BUSY (busy) • HAL_TIMEOUT (timeout) |
| Remarks | This API should be called when starting the SoC. If the SysTick is used as the source time base, overwriting SysTick_IRQHandler is required. Otherwise, polling APIs will be implemented in an infinite loop, resulting in application failures. |

2.3.3.2 hal_deinit

Table 2-20 hal_deinit API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_deinit(void) |
| Function Description | Deinitialize the System driver, and disable the tick. |
| Parameter | None |
| Return Value | HAL status |

| | |
|---------|--|
| Remarks | |
|---------|--|

2.3.3.3 hal_msp_init

Table 2-21 hal_msp_init API

| | |
|-----------------------------|---|
| Function Prototype | void hal_msp_init(void) |
| Function Description | Initialize the clocks, GPIOs, and interrupts related to the system driver. |
| Parameter | None |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite the API to initialize the GPIO pins and interrupts. |

2.3.3.4 hal_msp_deinit

Table 2-22 hal_msp_deinit API

| | |
|-----------------------------|---|
| Function Prototype | void hal_msp_deinit(void) |
| Function Description | Deinitialize the clocks, GPIOs, and interrupts related to the system driver. |
| Parameter | None |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite the API to deinitialize the GPIO pins and interrupts. |

2.3.3.5 hal_init_tick

Table 2-23 hal_init_tick API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_init_tick(uint32_t tick_priority) |
| Function Description | Initialize the default tick source time base, SysTick, and set the time base as 1 millisecond with a dedicated tick_priority. |
| Parameter | tick_priority: SysTick interrupt priority; value range: 0 to 15. |
| Return Value | HAL status |
| Remarks | This function is of weak type and is automatically called in the hal_init() function. If developers hope to use other tick sources, overwrite this function. |

2.3.3.6 hal_suspend_tick

Table 2-24 hal_suspend_tick API

| | |
|---------------------------|-----------------------------|
| Function Prototype | void hal_suspend_tick(void) |
|---------------------------|-----------------------------|

| | |
|-----------------------------|---|
| Function Description | Suspend the tick increment. |
| Parameter | None |
| Return Value | None |
| Remarks | The function is of weak type. Developers are required to overwrite the API in actual use. |

2.3.3.7 hal_resume_tick

Table 2-25 hal_resume_tick API

| | |
|-----------------------------|---|
| Function Prototype | void hal_resume_tick(void) |
| Function Description | Resume the tick increment. |
| Parameter | None |
| Return Value | None |
| Remarks | The function is of weak type. Developers are required to overwrite the API in actual use. |

2.3.3.8 hal_get_hal_version

Table 2-26 hal_get_hal_version API

| | |
|-----------------------------|--|
| Function Prototype | uint32_t hal_get_hal_version(void) |
| Function Description | Get the current HAL driver version. |
| Parameter | None |
| Return Value | HAL driver version: 0xAaBbCcDd; Aa is the major version, Bb is the minor version 1, Cc is the minor version 2, and Dd is the candidate release version. For example, 0x00000100 represents v0.0.1.0. |
| Remarks | |

2.4 HAL GPIO Generic Driver

2.4.1 GPIO Driver Functionalities

The HAL general-purpose input/output (GPIO) driver features the following functionalities:

- 32 GPIO pins work in input, output, and multiplexing modes.
- Interrupts of all GPIO pins can be triggered by four methods: low level, high level, rising edge, and falling edge.
- All GPIO pins have pull-up or pull-down resistors that can be enabled or disabled.
- Callback functions can be implemented after interrupts are triggered.

2.4.2 How to Use GPIO Driver

Developers can use the GPIO driver in the following scenarios:

1. Configure GPIO pins using `hal_gpio_init()`.
2. Configure the I/O mode using the **mode** member in the `gpio_init_t` structure.
3. Activate pull-up or pull-down resistors using the **pull** member in the `gpio_init_t` structure.
4. Enable I/O multiplexing using the **mux** member in the `gpio_init_t` structure.
5. Configure the GPIO interrupt priority by calling `hal_nvic_set_priority()`; enable GPIO interrupt handling by calling `hal_nvic_enable_irq()`.
6. Get the configured pin level in input mode through `hal_gpio_read_pin()`.
7. Set the configured pin level in output mode through `hal_gpio_write_pin()`, and reset the level through `hal_gpio_toggle_pin()`.

2.4.3 GPIO Driver Structures

2.4.3.1 gpio_init_t

The initialization structure `gpio_init_t` of the GPIO driver is defined below:

Table 2-27 `gpio_init_t` structure

| Data Field | Field Description | Value |
|---------------------------|---------------------------|--|
| <code>uint32_t pin</code> | GPIO pin to be configured | <p>This parameter can be any combination of the following values:</p> <ul style="list-style-type: none"> • <code>GPIO_PIN_0</code> (Pin 0) • <code>GPIO_PIN_1</code> (Pin 1) • <code>GPIO_PIN_2</code> (Pin 2) • <code>GPIO_PIN_3</code> (Pin 3) • <code>GPIO_PIN_4</code> (Pin 4) • <code>GPIO_PIN_5</code> (Pin 5) • <code>GPIO_PIN_6</code> (Pin 6) • <code>GPIO_PIN_7</code> (Pin 7) • <code>GPIO_PIN_8</code> (Pin 8) • <code>GPIO_PIN_9</code> (Pin 9) • <code>GPIO_PIN_10</code> (Pin 10) • <code>GPIO_PIN_11</code> (Pin 11) • <code>GPIO_PIN_12</code> (Pin 12) • <code>GPIO_PIN_13</code> (Pin 13) • <code>GPIO_PIN_14</code> (Pin 14) • <code>GPIO_PIN_15</code> (Pin 15) |

| Data Field | Field Description | Value |
|---------------|---|---|
| | | <ul style="list-style-type: none"> GPIO_PIN_ALL (Pins 0–15) |
| uint32_t mode | Operating mode of the selected pin | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> GPIO_MODE_INPUT (input mode) GPIO_MODE_OUTPUT (output mode) GPIO_MODE_MUX (multiplexing mode) GPIO_MODE_IT_RISING (external interrupts triggered by rising edge) GPIO_MODE_IT_FALLING (external interrupts triggered by falling edge) GPIO_MODE_IT_HIGH (external interrupts triggered by high level) GPIO_MODE_IT_LOW (external interrupts triggered by low level) |
| uint32_t pull | Enable/Disable the selected pin's pull-up resistor or pull-down resistor. | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> GPIO_NOPULL (disable internal pull-up/pull-down resistors) GPIO_PULLUP (enable internal pull-up resistors) GPIO_PULLDOWN (enable internal pull-down resistors) |
| uint32_t mux | Peripherals connected to the selected pins | See " Section 2.5 HAL GPIO Extension Driver ". |

2.4.4 GPIO Driver APIs

The GPIO driver APIs are listed in the table below:

Table 2-28 GPIO driver APIs

| API Type | API Name | Description |
|---------------------------------|-----------------------------|------------------------------------|
| Initialization | hal_gpio_init() | Initialize a specified GPIO pin. |
| | hal_gpio_deinit() | Deinitialize a specified GPIO pin. |
| I/O operation | hal_gpio_read_pin() | Read the input level of a pin. |
| | hal_gpio_write_pin() | Set the output level of a pin. |
| | hal_gpio_toggle_pin() | Toggle the output level of a pin. |
| Interrupt handling and callback | hal_gpio_exti_irq_handler() | Interrupt handler |
| | hal_gpio_exti_callback() | Interrupt callback |

The sections below elaborate on these APIs.

2.4.4.1 hal_gpio_init

Table 2-29 hal_gpio_init API

| | |
|-----------------------------|---|
| Function Prototype | void hal_gpio_init(gpio_regs_t* GPIOx, gpio_init_t *p_gpio_init) |
| Function Description | Initialize GPIO peripherals according to parameters of gpio_init_t . |
| Parameter | GPIOx: x can be 0 or 1 to select a GPIO peripheral in the GR551x family. p_gpio_init: pointer to variables of gpio_init_t . The variable contains the configuration information of a specified GPIO pin. |
| Return Value | None |
| Remarks | |

2.4.4.2 hal_gpio_deinit

Table 2-30 hal_gpio_deinit API

| | |
|-----------------------------|--|
| Function Prototype | void hal_gpio_deinit(gpio_regs_t* GPIOx, uint32_t gpio_pin) |
| Function Description | Deinitialize the GPIO peripheral registers to default reset values. |
| Parameter | GPIOx: x can be 0 or 1 to select a GPIO peripheral in the GR551x family. gpio_pin: specifies a pin bit to be written. This parameter can be any combination of the following values: <ul style="list-style-type: none"> • GPIO_PIN_0 (Pin 0) • GPIO_PIN_1 (Pin 1) • GPIO_PIN_2 (Pin 2) • GPIO_PIN_3 (Pin 3) • GPIO_PIN_4 (Pin 4) • GPIO_PIN_5 (Pin 5) • GPIO_PIN_6 (Pin 6) • GPIO_PIN_7 (Pin 7) • GPIO_PIN_8 (Pin 8) • GPIO_PIN_9 (Pin 9) • GPIO_PIN_10 (Pin 10) • GPIO_PIN_11 (Pin 11) • GPIO_PIN_12 (Pin 12) • GPIO_PIN_13 (Pin 13) • GPIO_PIN_14 (Pin 14) • GPIO_PIN_15 (Pin 15) • GPIO_PIN_ALL (Pins 0–15) |
| Return Value | None |

| | |
|---------|--|
| Remarks | |
|---------|--|

2.4.4.3 hal_gpio_read_pin

Table 2-31 hal_gpio_read_pin API

| | |
|-----------------------------|--|
| Function Prototype | gpio_pin_state_t hal_gpio_read_pin(gpio_regs_t* GPIOx, uint16_t gpio_pin) |
| Function Description | Read a specified input port pin. |
| Parameter | <p>GPIOx: x can be 0 or 1 to select a GPIO peripheral in the GR551x family.</p> <p>gpio_pin: specifies a pin bit to be read. This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • GPIO_PIN_0 (Pin 0) • GPIO_PIN_1 (Pin 1) • GPIO_PIN_2 (Pin 2) • GPIO_PIN_3 (Pin 3) • GPIO_PIN_4 (Pin 4) • GPIO_PIN_5 (Pin 5) • GPIO_PIN_6 (Pin 6) • GPIO_PIN_7 (Pin 7) • GPIO_PIN_8 (Pin 8) • GPIO_PIN_9 (Pin 9) • GPIO_PIN_10 (Pin 10) • GPIO_PIN_11 (Pin 11) • GPIO_PIN_12 (Pin 12) • GPIO_PIN_13 (Pin 13) • GPIO_PIN_14 (Pin 14) • GPIO_PIN_15 (Pin 15) |
| Return Value | Input port pin value |
| Remarks | |

2.4.4.4 hal_gpio_write_pin

Table 2-32 hal_gpio_write_pin API

| | |
|-----------------------------|--|
| Function Prototype | void hal_gpio_write_pin(gpio_regs_t* GPIOx, uint16_t gpio_pin, gpio_pin_state_t pin_state) |
| Function Description | Set or clear a selected data port bit. |
| Parameter | <p>GPIOx: x can be 0 or 1 to select a GPIO peripheral in the GR551x family.</p> <p>gpio_pin: specifies a pin bit to be written. This parameter can be any combination of the following values:</p> |

| | |
|---------------------|--|
| | <ul style="list-style-type: none"> • GPIO_PIN_0 (Pin 0) • GPIO_PIN_1 (Pin 1) • GPIO_PIN_2 (Pin 2) • GPIO_PIN_3 (Pin 3) • GPIO_PIN_4 (Pin 4) • GPIO_PIN_5 (Pin 5) • GPIO_PIN_6 (Pin 6) • GPIO_PIN_7 (Pin 7) • GPIO_PIN_8 (Pin 8) • GPIO_PIN_9 (Pin 9) • GPIO_PIN_10 (Pin 10) • GPIO_PIN_11 (Pin 11) • GPIO_PIN_12 (Pin 12) • GPIO_PIN_13 (Pin 13) • GPIO_PIN_14 (Pin 14) • GPIO_PIN_15 (Pin 15) • GPIO_PIN_ALL (Pins 0–15) <p>pin_state: specifies a value to be written to the selected pin. This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • GPIO_PIN_RESET (low level) • GPIO_PIN_SET (high level) |
| Return Value | None |
| Remarks | |

2.4.4.5 hal_gpio_toggle_pin

Table 2-33 hal_gpio_toggle_pin API

| | |
|-----------------------------|---|
| Function Prototype | void hal_gpio_toggle_pin(gpio_regs_t* GPIOx, uint16_t gpio_pin) |
| Function Description | Toggle a specified port pin. |
| Parameter | <p>GPIOx: x can be 0 or 1 to select a GPIO peripheral in the GR551x family.</p> <p>gpio_pin: specifies a pin bit to be toggled. This parameter can be any combination of the following values:</p> <ul style="list-style-type: none"> • GPIO_PIN_0 (Pin 0) • GPIO_PIN_1 (Pin 1) • GPIO_PIN_2 (Pin 2) |

| | |
|---------------------|--|
| | <ul style="list-style-type: none"> • GPIO_PIN_3 (Pin 3) • GPIO_PIN_4 (Pin 4) • GPIO_PIN_5 (Pin 5) • GPIO_PIN_6 (Pin 6) • GPIO_PIN_7 (Pin 7) • GPIO_PIN_8 (Pin 8) • GPIO_PIN_9 (Pin 9) • GPIO_PIN_10 (Pin 10) • GPIO_PIN_11 (Pin 11) • GPIO_PIN_12 (Pin 12) • GPIO_PIN_13 (Pin 13) • GPIO_PIN_14 (Pin 14) • GPIO_PIN_15 (Pin 15) • GPIO_PIN_ALL (Pins 0–15) |
| Return Value | None |
| Remarks | |

2.4.4.6 hal_gpio_exti_irq_handler

Table 2-34 hal_gpio_exti_irq_handler API

| | |
|-----------------------------|--|
| Function Prototype | void hal_gpio_exti_irq_handler(gpio_regs_t* GPIOx) |
| Function Description | Handle GPIO interrupt requests. |
| Parameter | GPIOx: x can be 0 or 1 to select a GPIO peripheral in the GR551x family. |
| Return Value | None |
| Remarks | |

2.4.4.7 hal_gpio_exti_callback

Table 2-35 hal_gpio_exti_callback API

| | |
|-----------------------------|--|
| Function Prototype | void hal_gpio_exti_callback(gpio_regs_t* GPIOx, uint16_t gpio_pin) |
| Function Description | GPIO interrupt callback function |
| Parameter | <p>GPIOx: x can be 0 or 1 to select a GPIO peripheral in the GR551x family.</p> <p>gpio_pin: the pin that triggers this interrupt. This parameter can be any combination of the following values:</p> <ul style="list-style-type: none"> • GPIO_PIN_0 (Pin 0) • GPIO_PIN_1 (Pin 1) |

| | |
|---------------------|--|
| | <ul style="list-style-type: none"> • GPIO_PIN_2 (Pin 2) • GPIO_PIN_3 (Pin 3) • GPIO_PIN_4 (Pin 4) • GPIO_PIN_5 (Pin 5) • GPIO_PIN_6 (Pin 6) • GPIO_PIN_7 (Pin 7) • GPIO_PIN_8 (Pin 8) • GPIO_PIN_9 (Pin 9) • GPIO_PIN_10 (Pin 10) • GPIO_PIN_11 (Pin 11) • GPIO_PIN_12 (Pin 12) • GPIO_PIN_13 (Pin 13) • GPIO_PIN_14 (Pin 14) • GPIO_PIN_15 (Pin 15) • GPIO_PIN_ALL (Pins 0–15) |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite this function before using it. |

2.5 HAL GPIO Extension Driver

The HAL GPIO extension driver defines macros for all GPIO pins in multiplexing mode depending on SoC series.

2.5.1 GPIO Driver Defines

2.5.1.1 GPIO Multiplexing Selection

- Common configurable item

| Macro | Description |
|-------------------|----------------------------------|
| GPIO_PIN_MUX_GPIO | Configure the pin as a GPIO pin. |

Note:

This macro applies to all pins.

- Configurable items for Pin 0 of GPIO0

| Macro | Description |
|-------------------------|---|
| GPIO0_PIN0_MUX_SWD_CLK | Configure the Pin 0 of GPIO0 as SWD_CLK. |
| GPIO0_PIN0_MUX_I2CO_SCL | Configure the Pin 0 of GPIO0 as I2CO_SCL. |

| Macro | Description |
|--------------------------|--|
| GPIO0_PIN0_MUX_I2C1_SCL | Configure the Pin 0 of GPIO0 as I2C1_SCL. |
| GPIO0_PIN0_MUX_UART1_RTS | Configure the Pin 0 of GPIO0 as UART1_RTS. |
| GPIO0_PIN0_MUX_UART0_TX | Configure the Pin 0 of GPIO0 as UART0_TX. |
| GPIO0_PIN0_MUX_UART1_TX | Configure the Pin 0 of GPIO0 as UART1_TX. |
| GPIO0_PIN0_MUX_UART0_RTS | Configure the Pin 0 of GPIO0 as UART0_RTS. |

- Configurable items for Pin 1 of GPIO0

| Macro | Description |
|--------------------------|--|
| GPIO0_PIN1_MUX_SWV_IO | Configure the Pin 1 of GPIO0 as SWV_IO. |
| GPIO0_PIN1_MUX_I2C0_SDA | Configure the Pin 1 of GPIO0 as I2C0_SDA. |
| GPIO0_PIN1_MUX_I2C1_SDA | Configure the Pin 1 of GPIO0 as I2C1_SDA. |
| GPIO0_PIN1_MUX_UART1_CTS | Configure the Pin 1 of GPIO0 as UART1_CTS. |
| GPIO0_PIN1_MUX_UART0_RX | Configure the Pin 1 of GPIO0 as UART0_RX. |
| GPIO0_PIN1_MUX_UART1_RX | Configure the Pin 1 of GPIO0 as UART1_RX. |
| GPIO0_PIN1_MUX_UART0_CTS | Configure the Pin 1 of GPIO0 as UART0_CTS. |

- Configurable items for Pin 2 of GPIO0

| Macro | Description |
|-----------------------------|---|
| GPIO0_PIN2_MUX_UART0_CTS | Configure the Pin 2 of GPIO0 as UART0_CTS. |
| GPIO0_PIN2_MUX_SIM_PRESENCE | Configure the Pin 2 of GPIO0 as SIM_PRESENCE. |
| GPIO0_PIN2_MUX_SWV | Configure the Pin 2 of GPIO0 as MUX_SWV. |
| GPIO0_PIN2_MUX_SPIS_CS_N | Configure the Pin 2 of GPIO0 as SPIS_CS_N. |
| GPIO0_PIN2_MUX_I2C0_SDA | Configure the Pin 2 of GPIO0 as I2C0_SDA. |
| GPIO0_PIN2_MUX_PWM0_A | Configure the Pin 2 of GPIO0 as PWM0 Channel A. |
| GPIO0_PIN2_MUX_FERP_TRIG | Configure the Pin 2 of GPIO0 as FERP_TRIG. |

- Configurable items for Pin 3 of GPIO0

| Macro | Description |
|----------------------------|---|
| GPIO0_PIN3_MUX_UART0_TX | Configure the Pin 3 of GPIO0 as UART0_TX. |
| GPIO0_PIN3_MUX_SIM_RST_N | Configure the Pin 3 of GPIO0 as SIM_RST_N. |
| GPIO0_PIN3_MUX_SPIM_CLK | Configure the Pin 3 of GPIO0 as SPIM_CLK. |
| GPIO0_PIN3_MUX_SPIS_CLK | Configure the Pin 3 of GPIO0 as SPIS_CLK. |
| GPIO0_PIN3_MUX_SPIM_CS1 | Configure the Pin 3 of GPIO0 as SPIM_CS1. |
| GPIO0_PIN3_MUX_PWM0_B | Configure the Pin 3 of GPIO0 as PWM0 Channel B. |
| GPIO0_PIN3_MUX_COEX_BLE_TX | Configure the Pin 3 of GPIO0 as COEX_BLE_TX. |

- Configurable items for Pin 4 of GPIO0

| Macro | Description |
|----------------------------|---|
| GPIO0_PIN4_MUX_UART0_RX | Configure the Pin 4 of GPIO0 as UART0_RX. |
| GPIO0_PIN4_MUX_SIM_IO | Configure the Pin 4 of GPIO0 as SIM_IO. |
| GPIO0_PIN4_MUX_SPIM_MOSI | Configure the Pin 4 of GPIO0 as SPIM_MOSI. |
| GPIO0_PIN4_MUX_SPIS_MISO | Configure the Pin 4 of GPIO0 as SPIS_MISO |
| GPIO0_PIN4_MUX_SPIM_CS0 | Configure the Pin 4 of GPIO0 as SPIM_CS0. |
| GPIO0_PIN4_MUX_PWM0_C | Configure the Pin 4 of GPIO0 as PWM0 Channel C. |
| GPIO0_PIN4_MUX_COEX_BLE_RX | Configure the Pin 4 of GPIO0 as COEX_BLE_RX. |

- Configurable items for Pin 5 of GPIO0

| Macro | Description |
|-----------------------------|---|
| GPIO0_PIN5_MUX_I2C0_SCL | Configure the Pin 5 of GPIO0 as I2C0_SCL |
| GPIO0_PIN5_MUX_UART0_RTS | Configure the Pin 5 of GPIO0 as UART0_RTS. |
| GPIO0_PIN5_MUX_SPIS_MOSI | Configure the Pin 5 of GPIO0 as SPIS_MOSI. |
| GPIO0_PIN5_MUX_SPIM_MISO | Configure the Pin 5 of GPIO0 as SPIM_MISO. |
| GPIO0_PIN5_MUX_SIM_CLK | Configure the Pin 5 of GPIO0 as SIM_CLK. |
| GPIO0_PIN5_MUX_COEX_WLAN_TX | Configure the Pin 5 of GPIO0 as COEX_WLAN_TX. |

- Configurable items for Pin 6 of GPIO0

| Macro | Description |
|-----------------------------|---|
| GPIO0_PIN6_MUX_I2C0_SDA | Configure the Pin 6 of GPIO0 as I2C0_SDA. |
| GPIO0_PIN6_MUX_I2SM_WS | Configure the Pin 6 of GPIO0 as I2SM_WS. |
| GPIO0_PIN6_MUX_I2SS_WS | Configure the Pin 6 of GPIO0 as I2SS_WS. |
| GPIO0_PIN6_MUX_SPIM_MOSI | Configure the Pin 6 of GPIO0 as SPIM_MOSI. |
| GPIO0_PIN6_MUX_SPIM_CS0 | Configure the Pin 6 of GPIO0 as SPIM_CS0. |
| GPIO0_PIN6_MUX_UART1_RX | Configure the Pin 6 of GPIO0 as UART1_RX. |
| GPIO0_PIN6_MUX_COEX_WLAN_RX | Configure the Pin 6 of GPIO0 as COEX_WLAN_RX. |

- Configurable items for Pin 7 of GPIO0

| Macro | Description |
|----------------------------|---|
| GPIO0_PIN7_MUX_I2SM_TX_SDO | Configure the Pin 7 of GPIO0 as I2SM_TX_SDO. |
| GPIO0_PIN7_MUX_I2SS_TX_SDO | Configure the Pin 7 of GPIO0 as I2SS_TX_SDO. |
| GPIO0_PIN7_MUX_SPIM_CS1 | Configure the Pin 7 of GPIO0 as SPIM_CS1. |
| GPIO0_PIN7_MUX_UART1_TX | Configure the Pin 7 of GPIO0 as UART1_TX. |
| GPIO0_PIN7_MUX_SPIM_CLK | Configure the Pin 7 of GPIO0 as SPIM_CLK. |
| GPIO0_PIN7_MUX_PWM1_A | Configure the Pin 7 of GPIO0 as PWM1 Channel A. |

| Macro | Description |
|------------------------------|--|
| GPIO0_PIN7_MUX_COEX_BLE_PROC | Configure the Pin 7 of GPIO0 as COEX_BLE_PROC. |

- Configurable items for Pin 8 of GPIO0

| Macro | Description |
|----------------------------|---|
| GPIO0_PIN8_MUX_XQSPIM_IO_0 | Configure the Pin 8 of GPIO0 as XQSPIM_IO_0. |
| GPIO0_PIN8_MUX_QSPIM1_IO_0 | Configure the Pin 8 of GPIO0 as QSPIM1_IO_0. |
| GPIO0_PIN8_MUX_I2C1_SDA | Configure the Pin 8 of GPIO0 as I2C1_SDA. |
| GPIO0_PIN8_MUX_UART1_RX | Configure the Pin 8 of GPIO0 as UART1_RX. |
| GPIO0_PIN8_MUX_PWM1_B | Configure the Pin 8 of GPIO0 as PWM1 Channel B. |

- Configurable items for Pin 9 of GPIO0

| Macro | Description |
|---------------------------|---|
| GPIO0_PIN9_MUX_XQSPIM_CLK | Configure the Pin 9 of GPIO0 as XQSPIM_CLK. |
| GPIO0_PIN9_MUX_QSPIM1_CLK | Configure the Pin 9 of GPIO0 as QSPIM1_CLK. |
| GPIO0_PIN9_MUX_I2C1_SCL | Configure the Pin 9 of GPIO0 as I2C1_SCL. |
| GPIO0_PIN9_MUX_UART1_TX | Configure the Pin 9 of GPIO0 as UART1_TX. |
| GPIO0_PIN9_MUX_PWM1_C | Configure the Pin 9 of GPIO0 as PWM1 Channel C. |

- Configurable items for Pin 10 of GPIO0

| Macro | Description |
|-----------------------------|--|
| GPIO0_PIN10_MUX_I2SM_RX_SDI | Configure the Pin 10 of GPIO0 as I2SM_RX_SDI. |
| GPIO0_PIN10_MUX_I2SS_RX_SDI | Configure the Pin 10 of GPIO0 as I2SS_RX_SDI. |
| GPIO0_PIN10_MUX_UART0_TX | Configure the Pin 10 of GPIO0 as UART0_TX. |
| GPIO0_PIN10_MUX_I2C0_SCL | Configure the Pin 10 of GPIO0 as I2C0_SCL. |
| GPIO0_PIN10_MUX_PWM1_B | Configure the Pin 10 of GPIO0 as PWM1 Channel B. |
| GPIO0_PIN10_MUX_COEX_BLE_TX | Configure the Pin 10 of GPIO0 as COEX_BLE_TX. |

- Configurable items for Pin 11 of GPIO0

| Macro | Description |
|---------------------------|--|
| GPIO0_PIN11_MUX_I2SM_SCLK | Configure the Pin 11 of GPIO0 as I2SM_SCLK. |
| GPIO0_PIN11_MUX_I2SS_SCLK | Configure the Pin 11 of GPIO0 as I2SS_SCLK. |
| GPIO0_PIN11_MUX_UART0_RX | Configure the Pin 11 of GPIO0 as UART0_RX. |
| GPIO0_PIN11_MUX_I2C0_SDA | Configure the Pin 11 of GPIO0 as I2C0_SDA. |
| GPIO0_PIN11_MUX_PWM1_C | Configure the Pin 11 of GPIO0 as PWM1 Channel C. |

- Configurable items for Pin 12 of GPIO0

| Macro | Description |
|------------------------------|--|
| GPIO0_PIN12_MUX_XQSPIM_IO_3 | Configure the Pin 12 of GPIO0 as XQSPIM_IO_3. |
| GPIO0_PIN12_MUX_SPIM_CLK | Configure the Pin 12 of GPIO0 as SPIM_CLK. |
| GPIO0_PIN12_MUX_QSPIM1_IO3 | Configure the Pin 12 of GPIO0 as QSPIM1_IO3. |
| GPIO0_PIN12_MUX_SIM_PRESENCE | Configure the Pin 12 of GPIO0 as SIM_PRESENCE. |
| GPIO0_PIN12_MUX_I2SM_WS | Configure the Pin 12 of GPIO0 as I2SM_WS. |
| GPIO0_PIN12_MUX_I2SS_WS | Configure the Pin 12 of GPIO0 as I2SS_WS. |
| GPIO0_PIN12_MUX_SPIS_CS | Configure the Pin 12 of GPIO0 as SPIS_CS. |

- Configurable items for Pin 13 of GPIO0

| Macro | Description |
|-----------------------------|---|
| GPIO0_PIN13_MUX_XQSPIM_IO_2 | Configure the Pin 13 of GPIO0 as XQSPIM_IO_2. |
| GPIO0_PIN13_MUX_SPIM_MOSI | Configure the Pin 13 of GPIO0 as SPIM_MOSI. |
| GPIO0_PIN13_MUX_QSPIM1_IO_2 | Configure the Pin 13 of GPIO0 as QSPIM1_IO_2. |
| GPIO0_PIN13_MUX_SIM_RST_N | Configure the Pin 13 of GPIO0 as SIM_RST_N. |
| GPIO0_PIN13_MUX_I2SM_TX_SDO | Configure the Pin 13 of GPIO0 as I2SM_TX_SDO. |
| GPIO0_PIN13_MUX_I2SS_TX_SDO | Configure the Pin 13 of GPIO0 as I2SS_TX_SDO. |
| GPIO0_PIN13_MUX_SPIS_CLK | Configure the Pin 13 of GPIO0 as SPIS_CLK. |

- Configurable items for Pin 14 of GPIO0

| Macro | Description |
|-----------------------------|---|
| GPIO0_PIN14_MUX_XQSPIM_IO_1 | Configure the Pin 14 of GPIO0 as XQSPIM_IO_1. |
| GPIO0_PIN14_MUX_SPIM_MISO | Configure the Pin 14 of GPIO0 as SPIM_MISO. |
| GPIO0_PIN14_MUX_QSPIM1_IO1 | Configure the Pin 14 of GPIO0 as QSPIM1_IO1. |
| GPIO0_PIN14_MUX_SIM_IO | Configure the Pin 14 of GPIO0 as SIM_IO. |
| GPIO0_PIN14_MUX_I2SM_RX_SDI | Configure the Pin 14 of GPIO0 as I2SM_RX_SDI. |
| GPIO0_PIN14_MUX_I2SS_RX_SDI | Configure the Pin 14 of GPIO0 as I2SS_RX_SDI. |
| GPIO0_PIN14_MUX_SPIS_MISO | Configure the Pin 14 of GPIO0 as SPIS_MISO. |

- Configurable items for Pin 15 of GPIO0

| Macro | Description |
|-----------------------------|---|
| GPIO0_PIN15_MUX_XQSPIM_CS_N | Configure the Pin 15 of GPIO0 as XQSPIM_CS_N. |
| GPIO0_PIN15_MUX_SPIM_CS0 | Configure the Pin 15 of GPIO0 as SPIM_CS0. |
| GPIO0_PIN15_MUX_QSPIM1_CS_N | Configure the Pin 15 of GPIO0 as QSPIM1_CS_N. |
| GPIO0_PIN15_MUX_SIM_CLK | Configure the Pin 15 of GPIO0 as SIM_CLK. |
| GPIO0_PIN15_MUX_I2SM_SCLK | Configure the Pin 15 of GPIO0 as I2SM_SCLK. |

| Macro | Description |
|---------------------------|---|
| GPIO0_PIN15_MUX_I2SS_SCLK | Configure the Pin 15 of GPIO0 as I2SS_SCLK. |
| GPIO0_PIN15_MUX_SPIS_MOSI | Configure the Pin 15 of GPIO0 as SPIS_MOSI. |

- Configurable items for Pin 0 of GPIO1

| Macro | Description |
|----------------------------|--|
| GPIO1_PIN0_MUX_ISO_SYNC | Configure the Pin 0 of GPIO1 as ISO_SYNC. |
| GPIO1_PIN0_MUX_SPIM_MISO | Configure the Pin 0 of GPIO1 as SPIM_MISO. |
| GPIO1_PIN0_MUX_QSPIM0_IO_1 | Configure the Pin 0 of GPIO1 as QSPIM0_IO_1. |
| GPIO1_PIN0_MUX_SPIS_MOSI | Configure the Pin 0 of GPIO1 as SPIS_MOSI. |
| GPIO1_PIN0_MUX_SIM_IO | Configure the Pin 0 of GPIO1 as SIM_IO. |
| GPIO1_PIN0_MUX_I2SM_RX_SDI | Configure the Pin 0 of GPIO1 as I2SM_RX_SDI. |
| GPIO1_PIN0_MUX_I2SS_RX_SDI | Configure the Pin 0 of GPIO1 as I2SS_RX_SDI. |

- Configurable items for Pin 1 of GPIO1

| Macro | Description |
|----------------------------|--|
| GPIO1_PIN1_MUX_SPIM_CS0 | Configure the Pin 1 of GPIO1 as SPIM_CS0. |
| GPIO1_PIN1_MUX_SPIS_CS | Configure the Pin 1 of GPIO1 as SPIS_CS. |
| GPIO1_PIN1_MUX_SIM_CLK | Configure the Pin 1 of GPIO1 as SIM_CLK. |
| GPIO1_PIN1_MUX_I2SM_SCLK | Configure the Pin 1 of GPIO1 as I2SM_SCLK. |
| GPIO1_PIN1_MUX_I2SS_SCLK | Configure the Pin 1 of GPIO1 as I2SS_SCLK. |
| GPIO1_PIN1_MUX_QSPIM0_IO_2 | Configure the Pin 1 of GPIO1 as QSPIM0_IO_2. |
| GPIO1_PIN1_MUX_COEX_BLE_RX | Configure the Pin 1 of GPIO1 as COEX_BLE_RX. |

- Configurable items for Pin 2 of GPIO1

| Macro | Description |
|-------------------------------|---|
| GPIO1_PIN2_MUX_QSPIM0_CS_N | Configure the Pin 2 of GPIO1 as QSPIM0_CS_N. |
| GPIO1_PIN2_MUX_XQSPIM_IO_CS_N | Configure the Pin 2 of GPIO1 as XQSPIM_IO_CS_N. |

- Configurable items for Pin 3 of GPIO1

| Macro | Description |
|----------------------------|--|
| GPIO1_PIN3_MUX_QSPIM0_IO_3 | Configure the Pin 3 of GPIO1 as QSPIM0_IO_3. |
| GPIO1_PIN3_MUX_XQSPIM_IO_3 | Configure the Pin 3 of GPIO1 as XQSPIM_IO_3. |

- Configurable items for Pin 4 of GPIO1

| Macro | Description |
|---------------------------|---|
| GPIO1_PIN4_MUX_QSPIM0_CLK | Configure the Pin 4 of GPIO1 as QSPIM0_CLK. |

| Macro | Description |
|---------------------------|---|
| GPIO1_PIN4_MUX_XQSPIM_CLK | Configure the Pin 4 of GPIO1 as XQSPIM_CLK. |

- Configurable items for Pin 5 of GPIO1

| Macro | Description |
|----------------------------|--|
| GPIO1_PIN5_MUX_QSPIM0_IO_2 | Configure the Pin 5 of GPIO1 as QSPIM0_IO_2. |
| GPIO1_PIN5_MUX_XQSPIM_IO_2 | Configure the Pin 5 of GPIO1 as XQSPIM_IO_2. |

- Configurable items for Pin 6 of GPIO1

| Macro | Description |
|----------------------------|--|
| GPIO1_PIN6_MUX_QSPIM0_IO_1 | Configure the Pin 6 of GPIO1 as QSPIM0_IO_1. |
| GPIO1_PIN6_MUX_XQSPIM_IO_1 | Configure the Pin 6 of GPIO1 as XQSPIM_IO_1. |

- Configurable items for Pin 7 of GPIO1

| Macro | Description |
|----------------------------|--|
| GPIO1_PIN7_MUX_QSPIM0_IO_0 | Configure the Pin 7 of GPIO1 as QSPIM0_IO_0. |
| GPIO1_PIN7_MUX_XQSPIM_IO_0 | Configure the Pin 7 of GPIO1 as XQSPIM_IO_0. |

- Configurable items for Pin 8 of GPIO1

| Macro | Description |
|-----------------------------|---|
| GPIO1_PIN8_MUX_SPIM_CLK | Configure the Pin 8 of GPIO1 as SPIM_CLK. |
| GPIO1_PIN8_MUX_SPIS_CLK | Configure the Pin 8 of GPIO1 as SPIS_CLK. |
| GPIO1_PIN8_MUX_SIM_PRESENCE | Configure the Pin 8 of GPIO1 as SIM_PRESENCE. |
| GPIO1_PIN8_MUX_I2SM_WS | Configure the Pin 8 of GPIO1 as I2SM_WS. |
| GPIO1_PIN8_MUX_I2SS_WS | Configure the Pin 8 of GPIO1 as I2SS_WS. |
| GPIO1_PIN8_MUX_QSPIM0_CLK | Configure the Pin 8 of GPIO1 as QSPIM0_CLK. |
| GPIO1_PIN8_MUX_COEX_WLAN_TX | Configure the Pin 8 of GPIO1 as COEX_WLAN_TX. |

- Configurable items for Pin 9 of GPIO1

| Macro | Description |
|------------------------------|--|
| GPIO1_PIN9_MUX_SPIM_MOSI | Configure the Pin 9 of GPIO1 as MUX_SPIM_MOSI. |
| GPIO1_PIN9_MUX_SPIS_MISO | Configure the Pin 9 of GPIO1 as SPIS_MISO. |
| GPIO1_PIN9_MUX_SIM_RST_N | Configure the Pin 9 of GPIO1 as SIM_RST_N. |
| GPIO1_PIN9_MUX_I2SM_TX_SD0 | Configure the Pin 9 of GPIO1 as I2SM_TX_SD0. |
| GPIO1_PIN9_MUX_I2SS_TX_SD0 | Configure the Pin 9 of GPIO1 as I2SS_TX_SD0. |
| GPIO1_PIN9_MUX_QSPIM0_IO_0 | Configure the Pin 9 of GPIO1 as QSPIM0_IO_0. |
| GPIO1_PIN9_MUX_COEX_BLE_PROC | Configure the Pin 9 of GPIO1 as COEX_BLE_PROC. |

- Configurable items for Pin 10 of GPIO1

| Macro | Description |
|--------------------------|--|
| GPIO1_PIN10_MUX_I2C1_SDA | Configure the Pin 10 of GPIO1 as I2C1_SDA. |
| GPIO1_PIN10_MUX_UART1_RX | Configure the Pin 10 of GPIO1 as UART1_RX. |
| GPIO1_PIN10_MUX_I2C0_SDA | Configure the Pin 10 of GPIO1 as I2C0_SDA. |
| GPIO1_PIN10_MUX_PWM0_C | Configure the Pin 10 of GPIO1 as PWM0 Channel C. |
| GPIO1_PIN10_MUX_PWM1_C | Configure the Pin 10 of GPIO1 as PWM1 Channel C. |
| GPIO1_PIN10_MUX_UART0_RX | Configure the Pin 10 of GPIO1 as UART0_RX. |

- Configurable items for Pin 11 of GPIO1

| Macro | Description |
|---------------------------|---|
| GPIO1_PIN11_MUX_UART1_RTS | Configure the Pin 11 of GPIO1 as UART1_RTS. |
| GPIO1_PIN11_MUX_UART0_RTS | Configure the Pin 11 of GPIO1 as UART0_RTS. |

- Configurable items for Pin 12 of GPIO1

| Macro | Description |
|---------------------------|---|
| GPIO1_PIN12_MUX_UART1_CTS | Configure the Pin 12 of GPIO1 as UART1_CTS. |
| GPIO1_PIN12_MUX_UART0_CTS | Configure the Pin 12 of GPIO1 as UART0_CTS. |

- Configurable items for Pin 14 of GPIO1

| Macro | Description |
|-----------------------------|--|
| GPIO1_PIN14_MUX_I2C1_SCL | Configure the Pin 14 of GPIO1 as I2C1_SCL. |
| GPIO1_PIN14_MUX_UART1_TX | Configure the Pin 14 of GPIO1 as UART1_TX. |
| GPIO1_PIN14_MUX_I2C0_SCL | Configure the Pin 14 of GPIO1 as I2C0_SCL. |
| GPIO1_PIN14_MUX_PWM0_B | Configure the Pin 14 of GPIO1 as PWM0 Channel B. |
| GPIO1_PIN14_MUX_PWM1_B | Configure the Pin 14 of GPIO1 as PWM1 Channel B. |
| GPIO1_PIN14_MUX_UART0_TX | Configure the Pin 14 of GPIO1 as UART0_TX. |
| GPIO1_PIN14_MUX_COEX_BLE_TX | Configure the Pin 14 of GPIO1 as COEX_BLE_TX. |

- Configurable items for Pin 15 of GPIO1

| Macro | Description |
|------------------------------|--|
| GPIO1_PIN15_MUX_SPIM_CS1 | Configure the Pin 15 of GPIO1 as SPIM_CS1. |
| GPIO1_PIN15_MUX_PWM0_A | Configure the Pin 15 of GPIO1 as PWM0 Channel A. |
| GPIO1_PIN15_MUX_PWM1_A | Configure the Pin 15 of GPIO1 as PWM1 Channel A. |
| GPIO1_PIN15_MUX_QSPIM0_IO_3 | Configure the Pin 15 of GPIO1 as QSPIM0_IO_3. |
| GPIO1_PIN15_MUX_COEX_WLAN_TX | Configure the Pin 15 of GPIO1 as COEX_WLAN_TX. |

2.6 HAL AON GPIO Generic Driver

2.6.1 AON GPIO Driver Functionalities

The HAL AON GPIO (Always-on GPIO) driver features the following functionalities:

- 8 pins work in input and output modes.
- Interrupts of all GPIO pins can be triggered by four methods: low level, high level, rising edge, and falling edge.
- The pin level remains at a certain value in deep sleep mode.
- AON_GPIO_5 outputs 2 MHz clock signals.
- Callback functions can be implemented after interrupts are triggered.

2.6.2 How to Use AON GPIO Driver

Developers can use the AON GPIO driver in the following scenarios:

1. Configure GPIO pins using `hal_aon_gpio_init()`.
 - Configure the I/O mode using the **mode** member in the `aon_gpio_init_t` structure.
 - Activate pull-up or pull-down resistors using the **pull** member in the `aon_gpio_init_t` structure.
 - Enable I/O multiplexing using the **mux** member in the `aon_gpio_init_t` structure.
 - Enable AON_GPIO interrupt handling by calling `hal_nvic_enable_irq()`.
2. If input interrupt of AON_GPIO is required, configure the AON_GPIO interrupt priority by calling `hal_nvic_set_priority()`; enable AON_GPIO interrupt handling by calling `hal_nvic_enable_irq()`.
3. Get the configured pin level in input mode through `hal_aon_gpio_read_pin()`.
4. Set the configured pin level in output mode through `hal_aon_gpio_write_pin()`, and reset the level through `hal_aon_gpio_toggle_pin()`.

2.6.3 AON GPIO Driver Structures

2.6.3.1 aon_gpio_init_t

The `spi_init_t` structure of the AON GPIO driver is defined below:

Table 2-36 aon_gpio_init_t structure

| Data Field | Field Description | Value |
|---------------------------|---|---|
| <code>uint32_t pin</code> | Specify an AON GPIO pin to be configured. | This parameter can be any combination of the following values: <ul style="list-style-type: none"> • AON_GPIO_PIN_0 • AON_GPIO_PIN_1 • AON_GPIO_PIN_2 |

| Data Field | Field Description | Value |
|---------------|---|--|
| | | <ul style="list-style-type: none"> AON_GPIO_PIN_3 AON_GPIO_PIN_4 AON_GPIO_PIN_5 AON_GPIO_PIN_6 AON_GPIO_PIN_7 AON_GPIO_PIN_ALL |
| uint32_t mode | Specify the operating mode of the selected pin. | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> AON_GPIO_MODE_INPUT (input mode) AON_GPIO_MODE_OUTPUT (output mode) AON_GPIO_MODE_IT_RISING (external interrupts triggered by rising edge) AON_GPIO_MODE_IT_FALLING (external interrupts triggered by falling edge) AON_GPIO_MODE_IT_HIGH (external interrupts triggered by high level) AON_GPIO_MODE_IT_LOW (external interrupts triggered by low level) |
| uint32_t pull | Activate the selected pin's pull-up resistor or pull-down resistor. | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> AON_GPIO_NOPULL (deactivate pull-up/pull-down resistors) AON_GPIO_PULLUP (activate pull-up resistors) AON_GPIO_PULLDOWN (activate pull-down resistors) |
| uint32_t mux | Peripherals connected to the selected pins | See " Section 2.7 HAL AON GPIO Extension Driver ". |

2.6.4 AON GPIO Driver APIs

The AON GPIO driver APIs are listed in the table below:

Table 2-37 AON GPIO driver APIs

| API Type | API Name | Description |
|---------------------------------|----------------------------|--|
| Initialization | hal_aon_gpio_init() | Initialize a specified AON GPIO pin. |
| | hal_aon_gpio_deinit() | Deinitialize a specified AON GPIO pin. |
| I/O operation | hal_aon_gpio_read_pin() | Read the input level of a pin. |
| | hal_aon_gpio_write_pin() | Set the output level of a pin. |
| | hal_aon_gpio_toggle_pin() | Toggle the output level of a pin. |
| Interrupt handling and callback | hal_aon_gpio_irq_handler() | Interrupt handler |

| API Type | API Name | Description |
|----------|-------------------------|--------------------|
| | hal_aon_gpio_callback() | Interrupt callback |

The sections below elaborate on these APIs.

2.6.4.1 hal_aon_gpio_init

Table 2-38 hal_aon_gpio_init API

| | |
|-----------------------------|---|
| Function Prototype | void hal_aon_gpio_init(aon_gpio_init_t *p_aon_gpio_init) |
| Function Description | Initialize AON GPIO peripherals according to parameters of aon_gpio_init_t. |
| Parameter | p_aon_gpio_init: pointer to variables of Section 2.6.3.1 aon_gpio_init_t . The variable contains the configuration information of a specified AON GPIO. |
| Return Value | None |
| Remarks | |

2.6.4.2 hal_aon_gpio_deinit

Table 2-39 hal_aon_gpio_deinit API

| | |
|-----------------------------|---|
| Function Prototype | void hal_aon_gpio_deinit(uint32_t aon_gpio_pin) |
| Function Description | Deinitialize the AON GPIO peripheral registers to default reset values. |
| Parameter | <p>aon_gpio_pin: specifies a pin bit to be written. This parameter can be any combination of the following values:</p> <ul style="list-style-type: none"> • AON_GPIO_PIN_0 • AON_GPIO_PIN_1 • AON_GPIO_PIN_2 • AON_GPIO_PIN_3 • AON_GPIO_PIN_4 • AON_GPIO_PIN_5 • AON_GPIO_PIN_6 • AON_GPIO_PIN_7 • AON_GPIO_PIN_ALL |
| Return Value | None |
| Remarks | |

2.6.4.3 hal_aon_gpio_read_pin

Table 2-40 hal_aon_gpio_read_pin API

| | |
|-----------------------------|--|
| Function Prototype | aon_gpio_pin_state_t hal_aon_gpio_read_pin(uint16_t aon_gpio_pin) |
| Function Description | Read the input level of a pin. |
| Parameter | <p>aon_gpio_pin: specifies a pin bit to be read. This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • AON_GPIO_PIN_0 • AON_GPIO_PIN_1 • AON_GPIO_PIN_2 • AON_GPIO_PIN_3 • AON_GPIO_PIN_4 • AON_GPIO_PIN_5 • AON_GPIO_PIN_6 • AON_GPIO_PIN_7 |
| Return Value | <p>The level of an input pin can be one of the following values:</p> <ul style="list-style-type: none"> • AON_GPIO_PIN_RESET (low level) • AON_GPIO_PIN_SET (high level) |
| Remarks | |

2.6.4.4 hal_aon_gpio_write_pin

Table 2-41 hal_aon_gpio_write_pin API

| | |
|-----------------------------|--|
| Function Prototype | void hal_aon_gpio_write_pin(uint16_t aon_gpio_pin, aon_gpio_pin_state_t pin_state) |
| Function Description | Set the output level of a pin. |
| Parameter | <p>aon_gpio_pin: specifies a pin. This parameter can be any combination of the following values:</p> <ul style="list-style-type: none"> • AON_GPIO_PIN_0 • AON_GPIO_PIN_1 • AON_GPIO_PIN_2 • AON_GPIO_PIN_3 • AON_GPIO_PIN_4 • AON_GPIO_PIN_5 • AON_GPIO_PIN_6 • AON_GPIO_PIN_7 • AON_GPIO_PIN_ALL <p>pin_state: specifies a pin level. This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • AON_GPIO_PIN_RESET (low level) |

| | |
|---------------------|---|
| | <ul style="list-style-type: none"> AON_GPIO_PIN_SET (high level) |
| Return Value | None |
| Remarks | |

2.6.4.5 hal_aon_gpio_toggle_pin

Table 2-42 hal_aon_gpio_toggle_pin API

| | |
|-----------------------------|---|
| Function Prototype | void hal_aon_gpio_toggle_pin(uint16_t aon_gpio_pin) |
| Function Description | Toggle the level of a pin. |
| Parameter | <p>aon_gpio_pin: specifies a pin to be toggled. This parameter can be any combination of the following values:</p> <ul style="list-style-type: none"> AON_GPIO_PIN_0 AON_GPIO_PIN_1 AON_GPIO_PIN_2 AON_GPIO_PIN_3 AON_GPIO_PIN_4 AON_GPIO_PIN_5 AON_GPIO_PIN_6 AON_GPIO_PIN_7 AON_GPIO_PIN_ALL |
| Return Value | None |
| Remarks | |

2.6.4.6 hal_aon_gpio_irq_handler

Table 2-43 hal_aon_gpio_irq_handler API

| | |
|-----------------------------|-------------------------------------|
| Function Prototype | void hal_aon_gpio_irq_handler(void) |
| Function Description | Handle AON GPIO interrupt requests. |
| Parameter | None |
| Return Value | None |
| Remarks | |

2.6.4.7 hal_aon_gpio_callback

Table 2-44 hal_aon_gpio_callback API

| | |
|---------------------------|---|
| Function Prototype | void hal_aon_gpio_callback(uint16_t aon_gpio_pin) |
|---------------------------|---|

| | |
|-----------------------------|--|
| Function Description | AON GPIO interrupt callback function |
| Parameter | <p>aon_gpio_pin: the pin that triggers this interrupt. This parameter can be any combination of the following values:</p> <ul style="list-style-type: none"> • AON_GPIO_PIN_0 • AON_GPIO_PIN_1 • AON_GPIO_PIN_2 • AON_GPIO_PIN_3 • AON_GPIO_PIN_4 • AON_GPIO_PIN_5 • AON_GPIO_PIN_6 • AON_GPIO_PIN_7 • AON_GPIO_PIN_ALL |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite this function before using it. |

2.7 HAL AON GPIO Extension Driver

The HAL AON GPIO extension driver defines macros for all AON GPIO pins in multiplexing mode depending on SoC series.

2.7.1 AON GPIO Driver Defines

2.7.1.1 AON GPIO Multiplexing Selection

- Common configurable item

| Macro | Description |
|-----------------------|----------------------------------|
| AON_GPIO_PIN_MUX_GPIO | Configure the pin as a GPIO pin. |

Note:

This macro applies to all pins.

- Configurable items for Pin 1 of AON GPIO

| Macro | Description |
|-------------------------------|---|
| AON_GPIO_PIN1_MUX_QSPIM0_CS_N | Configure the Pin 1 of AON GPIO as QSPIM0_CS_N. |
| AON_GPIO_PIN1_MUX_COEX_BLE_TX | Configure the Pin 1 of AON GPIO as COEX_BLE_TX. |

- Configurable items for Pin 2 of AON GPIO

| Macro | Description |
|---------------------------------|--|
| AON_GPIO_PIN2_MUX_SIM_PRESENCE | Configure the Pin 2 of AON GPIO as SIM_PRESENCE. |
| AON_GPIO_PIN2_MUX_QSPIM1_CS_N | Configure the Pin 2 of AON GPIO as QSPIM1_CS_N. |
| AON_GPIO_PIN2_MUX_I2S_WS | Configure the Pin 2 of AON GPIO as I2S_WS. |
| AON_GPIO_PIN2_MUX_I2S_S_WS | Configure the Pin 2 of AON GPIO as I2S_S_WS. |
| AON_GPIO_PIN2_MUX_PWM0_C | Configure the Pin 2 of AON GPIO as PWM0 Channel C. |
| AON_GPIO_PIN2_MUX_COEX_BLE_PROC | Configure the Pin 2 of AON GPIO as COEX_BLE_PROC. |

- Configurable items for Pin 3 of AON GPIO

| Macro | Description |
|--------------------------------|--|
| AON_GPIO_PIN3_MUX_SIM_RST_N | Configure the Pin 3 of AON GPIO as SIM_RST_N. |
| AON_GPIO_PIN3_MUX_QSPIM1_IO_0 | Configure the Pin 3 of AON GPIO as QSPIM1_IO_0. |
| AON_GPIO_PIN3_MUX_I2S_TX_SDO | Configure the Pin 3 of AON GPIO as I2S_TX_SDO. |
| AON_GPIO_PIN3_MUX_I2S_S_TX_SDO | Configure the Pin 3 of AON GPIO as I2S_S_TX_SDO. |
| AON_GPIO_PIN3_MUX_PWM1_A | Configure the Pin 3 of AON GPIO as PWM1 Channel A. |
| AON_GPIO_PIN3_MUX_COEX_WLAN_RX | Configure the Pin 3 of AON GPIO as COEX_WLAN_RX. |

- Configurable items for Pin 4 of AON GPIO

| Macro | Description |
|--------------------------------|--|
| AON_GPIO_PIN4_MUX_SIM_IO | Configure the Pin 4 of AON GPIO as SIM_IO. |
| AON_GPIO_PIN4_MUX_QSPIM1_IO_1 | Configure the Pin 4 of AON GPIO as QSPIM1_IO_1. |
| AON_GPIO_PIN4_MUX_I2S_RX_SDI | Configure the Pin 4 of AON GPIO as I2S_RX_SDI. |
| AON_GPIO_PIN4_MUX_I2S_S_RX_SDI | Configure the Pin 4 of AON GPIO as I2S_S_RX_SDI. |
| AON_GPIO_PIN4_MUX_PWM1_B | Configure the Pin 4 of AON GPIO as PWM1 Channel B. |
| AON_GPIO_PIN4_MUX_COEX_BLE_RX | Configure the Pin 4 of AON GPIO as COEX_BLE_RX. |

- Configurable items for Pin 5 of AON GPIO

| Macro | Description |
|--------------------------------|--|
| AON_GPIO_PIN5_MUX_SIM_CLK | Configure the Pin 5 of AON GPIO as SIM_CLK. |
| AON_GPIO_PIN5_MUX_QSPIM1_CLK | Configure the Pin 5 of AON GPIO as QSPIM1_CLK. |
| AON_GPIO_PIN5_MUX_I2S_SCLK | Configure the Pin 5 of AON GPIO as I2S_SCLK. |
| AON_GPIO_PIN5_MUX_I2S_S_SCLK | Configure the Pin 5 of AON GPIO as I2S_S_SCLK. |
| AON_GPIO_PIN5_MUX_PWM1_C | Configure the Pin 5 of AON GPIO as PWM1 Channel C. |
| AON_GPIO_PIN5_MUX_COEX_WLAN_TX | Configure the Pin 5 of AON GPIO as COEX_WLAN_TX. |

2.8 HAL MSIO Generic Driver

2.8.1 MSIO Driver Functionalities

The HAL Mixed Signal Input/Output (MSIO) driver features the following functionalities:

- Five I/O pins work in input and output modes.
- The MSIOs can be configured as Analog-to-digital Converter (ADC) input.

2.8.2 How to Use MSIO Driver

Developers can use the MSIO driver in the following scenarios:

1. Configure MSIO pins using `hal_msio_init()`.
 - Configure the I/O direction using the **direction** member in the `msio_init_t` structure.
 - Configure the I/O mode using the **mode** member in the `msio_init_t` structure.
 - Activate pull-up or pull-down resistors using the **pull** member in the `msio_init_t` structure.
 - Enable I/O multiplexing using the **mux** member in the `msio_init_t` structure.
2. If an MSIO pin is used as ADC input, configure the MSIO pin mode as `MSIO_MODE_ANALOG`.
3. Get the configured pin level in input mode through `hal_msio_read_pin()`.
4. Set the configured pin level in output mode through `hal_msio_write_pin()`, and reset the level through `hal_msio_toggle_pin()`.

2.8.3 MSIO Driver Structures

2.8.3.1 `msio_init_t`

The `msio_init_t` structure of the MSIO driver is defined below:

Table 2-45 `msio_init_t` structure

| Data Field | Field Description | Value |
|---------------------------------|---|---|
| <code>uint32_t pin</code> | Specify an MSIO pin to be configured. | This parameter can be any combination of the following values: <ul style="list-style-type: none"> • <code>MSIO_PIN_0</code> • <code>MSIO_PIN_1</code> • <code>MSIO_PIN_2</code> • <code>MSIO_PIN_3</code> • <code>MSIO_PIN_4</code> • <code>MSIO_PIN_ALL</code> |
| <code>uint32_t direction</code> | Specify the direction (input/output) of the selected pin. | This parameter can be one of the following values: <ul style="list-style-type: none"> • <code>MSIO_DIRECTION_NONE</code> (disable input and output) • <code>MSIO_DIRECTION_INPUT</code> (enable input) |

| Data Field | Field Description | Value |
|---------------|---|---|
| | | <ul style="list-style-type: none"> MSIO_DIRECTION_OUTPUT (enable output) MSIO_DIRECTION_INOUT (enable input and output) |
| uint32_t mode | Specify the operating mode of the selected pin. | This parameter can be one of the following values: <ul style="list-style-type: none"> MSIO_MODE_ANALOG (analog mode) MSIO_MODE_DIGITAL (digital mode) |
| uint32_t pull | Activate the selected pin's pull-up resistor or pull-down resistor. | This parameter can be one of the following values: <ul style="list-style-type: none"> MSIO_NOPULL (deactivate pull-up/pull-down resistors) MSIO_PULLUP (activate pull-up resistors) MSIO_PULLDOWN (activate pull-down resistors) |
| uint32_t mux | Peripherals connected to the selected pins | For details, see " Section 2.9 HAL MSIO Extension Driver ". |

2.8.4 MSIO Driver APIs

The MSIO driver APIs are listed in the table below:

Table 2-46 MSIO driver APIs

| API Type | API Name | Description |
|----------------|-----------------------|------------------------------------|
| Initialization | hal_msio_init() | Initialize a specified MSIO pin. |
| | hal_msio_deinit() | Deinitialize a specified MSIO pin. |
| I/O operation | hal_msio_read_pin() | Read the input level of a pin. |
| | hal_msio_write_pin() | Set the output level of a pin. |
| | hal_msio_toggle_pin() | Toggle the output level of a pin. |

The sections below elaborate on these APIs.

2.8.4.1 hal_msio_init

Table 2-47 hal_msio_init API

| | |
|-----------------------------|---|
| Function Prototype | void hal_msio_init(msio_init_t *p_msio_init) |
| Function Description | Initialize the MSIO peripheral according to parameters of msio_init_t. |
| Parameter | p_msio_init: pointer to variables of msio_init_t . The variable contains the configuration information of a specified MSIO. |
| Return Value | None |
| Remarks | |

2.8.4.2 hal_msio_deinit

Table 2-48 hal_msio_deinit API

| | |
|-----------------------------|---|
| Function Prototype | void hal_msio_deinit(uint32_t msio_pin) |
| Function Description | Deinitialize the MSIO peripheral registers to default reset values. |
| Parameter | <p>msio_pin: specifies a pin bit to be written. This parameter can be any combination of the following values:</p> <ul style="list-style-type: none"> • MSIO_PIN_0 • MSIO_PIN_1 • MSIO_PIN_2 • MSIO_PIN_3 • MSIO_PIN_4 • MSIO_PIN_ALL |
| Return Value | None |
| Remarks | |

2.8.4.3 hal_msio_read_pin

Table 2-49 hal_msio_read_pin API

| | |
|-----------------------------|--|
| Function Prototype | msio_pin_state_t hal_msio_read_pin(uint16_t msio_pin) |
| Function Description | Read the input level of a pin. |
| Parameter | <p>msio_pin: specifies a pin bit to be read. This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • MSIO_PIN_0 • MSIO_PIN_1 • MSIO_PIN_2 • MSIO_PIN_3 • MSIO_PIN_4 |
| Return Value | <p>The level of an input pin can be one of the following values:</p> <ul style="list-style-type: none"> • MSIO_PIN_RESET (low level) • MSIO_PIN_SET (high level) |
| Remarks | |

2.8.4.4 hal_msio_write_pin

Table 2-50 hal_msio_write_pin API

| | |
|-----------------------------|--|
| Function Prototype | void hal_msio_write_pin(uint16_t msio_pin, msio_pin_state_t pin_state) |
| Function Description | Set the output level of a pin. |

| | |
|---------------------|---|
| Parameter | <p>msio_pin: specifies a pin. This parameter can be any combination of the following values:</p> <ul style="list-style-type: none"> • MSIO_PIN_0 • MSIO_PIN_1 • MSIO_PIN_2 • MSIO_PIN_3 • MSIO_PIN_4 • MSIO_PIN_ALL <p>pin_state: specifies a pin level. This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • MSIO_PIN_RESET (low level) • MSIO_PIN_SET (high level) |
| Return Value | None |
| Remarks | |

2.8.4.5 hal_msio_toggle_pin

Table 2-51 hal_msio_toggle_pin API

| | |
|-----------------------------|---|
| Function Prototype | void hal_msio_toggle_pin(uint16_t msio_pin) |
| Function Description | Toggle the level of a pin. |
| Parameter | <p>msio_pin: specifies a pin to be toggled. This parameter can be any combination of the following values:</p> <ul style="list-style-type: none"> • MSIO_PIN_0 • MSIO_PIN_1 • MSIO_PIN_2 • MSIO_PIN_3 • MSIO_PIN_4 • MSIO_PIN_ALL |
| Return Value | None |
| Remarks | |

2.9 HAL MSIO Extension Driver

The HAL MSIO extension driver defines macros for all MSIO pins in multiplexing mode depending on SoC series.

2.9.1 MSIO Driver Defines

2.9.1.1 MSIO Multiplexing Selection

- Common configurable item

| Macro | Description |
|-------------------|----------------------------------|
| MSIO_PIN_MUX_GPIO | Configure the pin as a GPIO pin. |

 **Note:**

This macro applies to all pins.

- Configurable items for Pin 0 of MSIO

| Macro | Description |
|------------------------|--|
| MSIO_PIN0_MUX_PWM0_A | Configure the Pin 0 of MSIO as PWM0 Channel A. |
| MSIO_PIN0_MUX_UART0_TX | Configure the Pin 0 of MSIO as UART0_TX. |
| MSIO_PIN0_MUX_UART1_TX | Configure the Pin 0 of MSIO as UART1_TX. |
| MSIO_PIN0_MUX_I2C0_SCL | Configure the Pin 0 of MSIO as I2C0_SCL. |
| MSIO_PIN0_MUX_I2C1_SCL | Configure the Pin 0 of MSIO as I2C1_SCL. |

- Configurable items for Pin 1 of MSIO

| Macro | Description |
|------------------------|--|
| MSIO_PIN1_MUX_PWM0_B | Configure the Pin 1 of MSIO as PWM0 Channel B. |
| MSIO_PIN1_MUX_UART0_RX | Configure the Pin 1 of MSIO as UART0_RX. |
| MSIO_PIN1_MUX_UART1_RX | Configure the Pin 1 of MSIO as UART1_RX. |
| MSIO_PIN1_MUX_I2C0_SDA | Configure the Pin 1 of MSIO as I2C0_SDA. |
| MSIO_PIN1_MUX_I2C1_SDA | Configure the Pin 1 of MSIO as I2C1_SDA. |

- Configurable item for Pin 2 of MSIO

| Macro | Description |
|----------------------|--|
| MSIO_PIN2_MUX_PWM0_C | Configure the Pin 2 of MSIO as PWM0 Channel C. |

- Configurable items for Pin 3 of MSIO

| Macro | Description |
|-------------------------|--|
| MSIO_PIN3_MUX_PWM1_A | Configure the Pin 3 of MSIO as PWM1 Channel A. |
| MSIO_PIN3_MUX_UART0_RTS | Configure the Pin 3 of MSIO as UART0_RTS. |
| MSIO_PIN3_MUX_UART1_RTS | Configure the Pin 3 of MSIO as UART1_RTS. |
| MSIO_PIN3_MUX_I2C0_SCL | Configure the Pin 3 of MSIO as I2C0_SCL. |
| MSIO_PIN3_MUX_I2C1_SCL | Configure the Pin 3 of MSIO as I2C1_SCL. |

- Configurable items for Pin 4 of MSIO

| Macro | Description |
|----------------------|--|
| MSIO_PIN4_MUX_PWM1_B | Configure the Pin 4 of MSIO as PWM1 Channel B. |

| Macro | Description |
|-------------------------|---|
| MSIO_PIN4_MUX_UART0_CTS | Configure the Pin 4 of MSIO as UART0_CTS. |
| MSIO_PIN4_MUX_UART1_CTS | Configure the Pin 4 of MSIO as UART1_CTS. |
| MSIO_PIN4_MUX_I2C0_SDA | Configure the Pin 4 of MSIO as I2C0_SDA. |
| MSIO_PIN4_MUX_I2C1_SDA | Configure the Pin 4 of MSIO as I2C1_SDA. |

2.10 HAL ADC Generic Driver

2.10.1 ADC Driver Functionalities

The HAL ADC driver features the following functionalities:

- Two input modes: single-ended and differential
- Up to 1 Msps sampling rate
- Six clock rates: 1 MHz, 1.6 MHz, 2 MHz, 4 MHz, 8 MHz, and 16 MHz
- 13-bit sampling resolution
- Configurable internal reference voltages: 0.85 V, 1.28 V, and 1.6 V
- External reference voltage as input supply
- Ability to capture ADC samples using DMA, unburdening the MCU

2.10.2 How to Use ADC Driver

Developers can use HAL ADC driver in the following scenarios:

1. Declare an `adc_handle_t` handle structure, for example: `adc_handle_t adc_handle`.
2. Initialize the ADC low-level resources by overwriting `hal_adc_msp_init()`:
 - (1). ADC pin configuration: Configure the MSIO mode as `MSIO_MODE_ANALOG` (analog mode) by calling `hal_msio_init()`, and specify an MSIO pin to be configured as an analog I/O pin.
 - (2). If you need to use DMA process, `hal_adc_start_dma()`, you need to configure DMA:
 - Declare a DMA channel for ADC channels.
 - Declare a DMA handle structure for ADC channels, for example: `dma_handle_t hdma`.
 - Configure parameters in a DMA handle, for example, data exchange channels.
 - Associate the initial DMA handle with `p_dma` pointer of `adc_handle`.
 - Configure the DMA interrupt priority, and enable NVIC interrupts for DMA.
3. Configure parameters, such as reference voltage, in the init structure in `adc_handle`.
4. Initialize ADC registers by calling `hal_adc_init()`.

Note:

If an external power supply (ADC_REF_SRC_IOx, of which x can be a value from 0 to 3) is used as reference voltage, the input voltage ranges from 0.7 V to 1.9 V.

2.10.3 ADC Driver Structures

2.10.3.1 adc_init_t

The initialization structure `adc_init_t` of ADC driver is defined below:

Table 2-52 `adc_init_t` structure

| Data Field | Field Description | Value |
|----------------------------------|-----------------------|---|
| <code>uint32_t channel_p</code> | Input for Channel P | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • <code>ADC_INPUT_SRC_IO0</code> (MSIO0 input) • <code>ADC_INPUT_SRC_IO1</code> (MSIO1 input) • <code>ADC_INPUT_SRC_IO2</code> (MSIO2 input) • <code>ADC_INPUT_SRC_IO3</code> (MSIO3 input) • <code>ADC_INPUT_SRC_IO4</code> (MSIO4 input) • <code>ADC_INPUT_SRC_TMP</code> (temperature sensor input) • <code>ADC_INPUT_SRC_BAT</code> (battery voltage input) • <code>ADC_INPUT_SRC_REF</code> (reference voltage input) |
| <code>uint32_t channel_n</code> | Input for Channel N | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • <code>ADC_INPUT_SRC_IO0</code> (MSIO0 input) • <code>ADC_INPUT_SRC_IO1</code> (MSIO1 input) • <code>ADC_INPUT_SRC_IO2</code> (MSIO2 input) • <code>ADC_INPUT_SRC_IO3</code> (MSIO3 input) • <code>ADC_INPUT_SRC_IO4</code> (MSIO4 input) • <code>ADC_INPUT_SRC_TMP</code> (temperature sensor input) • <code>ADC_INPUT_SRC_BAT</code> (battery voltage input) • <code>ADC_INPUT_SRC_REF</code> (reference voltage input) |
| <code>uint32_t input_mode</code> | Sampling mode | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • <code>ADC_INPUT_SINGLE</code> (single-ended input mode) • <code>ADC_INPUT_DIFFERENTIAL</code> (differential input mode) |
| <code>uint32_t ref_source</code> | Reference source type | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • <code>ADC_REF_SRC_BUF_INT</code> (internal buffered reference source) • <code>ADC_REF_SRC_IO0</code> (MSIO0 input voltage) |

| Data Field | Field Description | Value |
|-------------------|----------------------------|---|
| | | <ul style="list-style-type: none"> • ADC_REF_SRC_IO1 (MSIO1 input voltage) • ADC_REF_SRC_IO2 (MSIO2 input voltage) • ADC_REF_SRC_IO3 (MSIO3 input voltage) |
| uin32_t ref_value | Internal reference voltage | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • ADC_REF_VALUE_0P8 (0.85 V) • ADC_REF_VALUE_1P2 (1.28 V) • ADC_REF_VALUE_1P6 (1.6 V) <p>Note: The external input signal range: 0 to (2 x ref_value). You can set this value based on actual requirements.</p> |
| uin32_t clock | Sampling clock | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • ADC_CLK_16M (16 MHz clock) • ADC_CLK_1P6M (1.6 MHz clock) • ADC_CLK_8M (8 MHz clock) • ADC_CLK_4M (4 MHz clock) • ADC_CLK_2M (2 MHz clock) • ADC_CLK_1M (1 MHz clock) |

2.10.3.2 adc_handle_t

The adc_handle_t structure of ADC driver is defined below:

Table 2-53 adc_handle_t structure

| Data Field | Field Description | Value |
|--------------------------|---|-------|
| adc_init_t init | Initialization structure (see " Section 2.10.3.1 adc_init_t ".) | N/A |
| uint16_t *p_buffer | Pointer to data RX buffer (managed by ADC driver and initialization by developers not required) | N/A |
| __IO uint32_t buff_size | Data RX buffer size (managed by ADC driver and initialization by developers not required) | N/A |
| __IO uint32_t buff_count | Data RX buffer count (managed by ADC driver and initialization by developers not required) | N/A |

| Data Field | Field Description | Value |
|----------------------------|--|--|
| dma_handle_t *p_dma | Pointer to <code>dma_handle_t</code> structure of DMA handle for data RX channels | N/A |
| __IO hal_lock_t lock | ADC lock (managed by ADC driver and initialization by developers not required) | N/A |
| __IO hal_adc_state_t state | ADC operating state (initialization by developers not required) | This parameter can be one of the following values: <ul style="list-style-type: none"> HAL_ADC_STATE_RESET (not initialized) HAL_ADC_STATE_READY (initialized and ready for use) HAL_ADC_STATE_BUSY (busy) HAL_ADC_STATE_ERROR (error) |
| __IO uint32_t error_code | ADC error code (initialization by developers not required) | This parameter can be one of the following values: <ul style="list-style-type: none"> HAL_ADC_ERROR_NONE (no error) HAL_ADC_ERROR_TIMEOUT (timeout) HAL_ADC_ERROR_DMA (DMA transfer error) HAL_ADC_ERROR_INVALID_PARAM (invalid parameter) |
| uint32_t retention[2] | ADC register information (managed by ADC driver and initialization by developers not required) | N/A |

2.10.4 ADC Driver APIs

The ADC driver APIs are listed in the table below:

Table 2-54 ADC driver APIs

| API Type | API Name | Description |
|----------------|-------------------------------|---|
| Initialization | hal_adc_init() | Initialize the ADC peripheral and configure reference voltage. |
| | hal_adc_deinit() | Deinitialize the ADC peripheral. |
| | hal_adc_msp_init() | Initialize MSIO pins, NVIC interrupts, and DMA channels used by the ADC peripheral. |
| | hal_adc_msp_deinit() | Deinitialize MSIO pins, NVIC interrupts, and DMA channels used by the ADC peripheral. |
| I/O operation | hal_adc_poll_for_conversion() | Take data samples in polling mode. |
| | hal_adc_start_dma() | Take data samples in DMA (non-polling) mode. |

| API Type | API Name | Description |
|---------------------------------|------------------------------|---|
| | hal_adc_stop_dma() | Abort data sampling in DMA (non-polling) mode. |
| Interrupt handling and callback | hal_adc_conv_cplt_callback() | Sampling in non-polling mode complete callback, defined by developers |
| State and error | hal_adc_get_state() | Get the driver operating state. |
| | hal_adc_get_error() | Get error code. |
| Control | hal_adc_set_dma_threshold() | Set a DMA threshold. |
| | hal_adc_get_dma_threshold() | Get a DMA threshold. |
| Sleep | hal_adc_suspend_reg() | Suspend registers related to ADC configuration in sleep mode. |
| | hal_adc_resume_reg() | Resume registers related to ADC configuration during wakeup. |

The sections below elaborate on these APIs.

2.10.4.1 hal_adc_init

Table 2-55 hal_adc_init API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_adc_init(adc_handle_t *p_adc) |
| Function Description | Initialize the ADC peripheral and related handles according to parameters of adc_init_t . |
| Parameter | p_adc: pointer to variables of adc_handle_t . The variable contains the configuration information of a specified ADC. |
| Return Value | HAL status |
| Remarks | |

2.10.4.2 hal_adc_deinit

Table 2-56 hal_adc_deinit API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_adc_deinit(adc_handle_t *p_adc) |
| Function Description | Deinitialize the ADC peripheral. |
| Parameter | p_adc: pointer to variables of adc_handle_t . The variable contains the configuration information of a specified ADC. |
| Return Value | HAL status |
| Remarks | |

2.10.4.3 hal_adc_msp_init

Table 2-57 hal_adc_msp_init API

| | |
|-----------------------------|--|
| Function Prototype | void hal_adc_msp_init(adc_handle_t *p_adc) |
| Function Description | Initialize the MSIO pins, NVIC interrupts, and DMA channels used by ADC. |
| Parameter | p_adc: pointer to variables of adc_handle_t . The variable contains the configuration information of a specified ADC. |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite the API to initialize MSIO pin selection, NVIC interrupts, and DMA channels. |

2.10.4.4 hal_adc_msp_deinit

Table 2-58 hal_adc_msp_deinit API

| | |
|-----------------------------|--|
| Function Prototype | void hal_adc_msp_deinit(adc_handle_t *p_adc) |
| Function Description | Deinitialize the MSIO pins, NVIC interrupts, and DMA channels used by ADC. |
| Parameter | p_adc: pointer to variables of adc_handle_t . The variable contains the configuration information of a specified ADC. |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite the API to deinitialize MSIO pin selection, NVIC interrupts, and DMA channels. |

2.10.4.5 hal_adc_poll_for_conversion

Table 2-59 hal_adc_poll_for_conversion API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_adc_poll_for_conversion(adc_handle_t *p_adc, uint16_t *p_data, uint32_t length) |
| Function Description | Enable ADC conversion, and read the converted data in polling mode. |
| Parameter | p_adc: pointer to variables of adc_handle_t . The variable contains the configuration information of a specified ADC. p_data: pointer to the data buffer that stores the ADC conversion results length: data buffer length |
| Return Value | HAL status |
| Remarks | When it returns HAL_ERROR, you can call hal_adc_get_error to retrieve the error code. |

2.10.4.6 hal_adc_start_dma

Table 2-60 hal_adc_start_dma API

| | |
|---------------------------|--|
| Function Prototype | hal_status_t hal_adc_start_dma(adc_handle_t *p_adc, uint16_t *p_data, uint32_t length) |
|---------------------------|--|

| | |
|-----------------------------|--|
| Function Description | Enable ADC conversion, and read the converted data in DMA mode. |
| Parameter | p_adc: pointer to variables of adc_handle_t . The variable contains the configuration information of a specified ADC. p_data: pointer to the data buffer that stores the ADC conversion results length: data buffer length |
| Return Value | HAL status |
| Remarks | When it returns HAL_ERROR, you can call hal_adc_get_error() to retrieve the error code. |

2.10.4.7 hal_adc_stop_dma

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_adc_stop_dma(adc_handle_t *p_adc) |
| Function Description | Abort ongoing ADC conversion, and read the converted data in DMA mode. |
| Parameter | p_adc: pointer to variables of adc_handle_t . The variable contains the configuration information of a specified ADC. |
| Return Value | HAL status |
| Remarks | This function only aborts ADC conversion that is enabled in DMA mode. Follow the steps below to abort such a conversion: <ol style="list-style-type: none"> 1. Disable ADC clock to abort conversion. 2. Abort DMA transfer by calling hal_dma_abort. 3. Set the handle state to READY. |

Table 2-61 hal_adc_stop_dma API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_adc_stop_dma (adc_handle_t *p_adc) |
| Function Description | Abort ongoing ADC conversion, and read the converted data in polling mode. |
| Parameter | p_adc: pointer to variables of adc_handle_t . The variable contains the configuration information of a specified ADC. |
| Return Value | HAL status |
| Remarks | This function only aborts ADC conversion that is enabled in DMA mode. Follow the steps below to abort such a conversion: <ul style="list-style-type: none"> • Disable ADC clock to abort conversion. • Abort DMA transfer by calling hal_dma_abort. • Set the handle state to READY. |

2.10.4.8 hal_adc_conv_cplt_callback

Table 2-62 hal_adc_conv_cplt_callback API

| | |
|---------------------------|--|
| Function Prototype | void hal_adc_conv_cplt_callback(adc_handle_t *p_adc) |
|---------------------------|--|

| | |
|-----------------------------|---|
| Function Description | Conversion complete callback function |
| Parameter | p_adc: pointer to variables of adc_handle_t . The variable contains the configuration information of a specified ADC. |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite the API to complete operations after sampling. |

2.10.4.9 hal_adc_get_state

Table 2-63 hal_adc_get_state API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_adc_get_state(adc_handle_t *p_adc) |
| Function Description | Return the ADC handle status. |
| Parameter | p_adc: pointer to variables of adc_handle_t . The variable contains the configuration information of a specified ADC. |
| Return Value | None |
| Remarks | The ADC state. This parameter can be one of the following values: <ul style="list-style-type: none"> • HAL_ADC_STATE_RESET (not initialized) • HAL_ADC_STATE_READY (initialized and ready for use) • HAL_ADC_STATE_BUSY (busy) • HAL_ADC_STATE_ERROR (error) |

2.10.4.10 hal_adc_get_error

Table 2-64 hal_adc_get_error API

| | |
|-----------------------------|--|
| Function Prototype | uint32_t hal_adc_get_error(adc_handle_t *p_adc) |
| Function Description | Return the ADC handle error code. |
| Parameter | p_adc: pointer to variables of adc_handle_t . The variable contains the configuration information of a specified ADC. |
| Return Value | ADC error code. This parameter can be one of the following values: <ul style="list-style-type: none"> • HAL_ADC_ERROR_NONE (no error) • HAL_ADC_ERROR_DMA (DMA transfer error) • HAL_ADC_ERROR_INVALID_PARAM (invalid parameter) • HAL_ADC_ERROR_TIMEOUT (timeout) |
| Remarks | |

2.10.4.11 hal_adc_set_dma_threshold

Table 2-65 hal_adc_set_dma_threshold API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_adc_set_dma_threshold(adc_handle_t *p_adc, uint32_t threshold) |
| Function Description | Set a FIFO threshold that triggers DMA transfer. |
| Parameter | p_adc: pointer to variables of adc_handle_t . The variable contains the configuration information of a specified ADC. threshold: FIFO trigger threshold (range: 0 to 64) |
| Return Value | HAL status |
| Remarks | |

2.10.4.12 hal_adc_get_dma_threshold

Table 2-66 hal_adc_get_dma_threshold API

| | |
|-----------------------------|---|
| Function Prototype | uint32_t hal_adc_get_dma_threshold(adc_handle_t *p_adc) |
| Function Description | Get the FIFO threshold that triggers DMA transfer. |
| Parameter | p_adc: pointer to variables of adc_handle_t . The variable contains the configuration information of a specified ADC. |
| Return Value | None |
| Remarks | The FIFO threshold ranges from 0 to 64. |

2.10.4.13 hal_adc_suspend_reg

Table 2-67 hal_adc_suspend_reg API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_adc_suspend_reg(adc_handle_t *p_adc) |
| Function Description | Suspend registers related to ADC configuration in sleep mode. |
| Parameter | p_adc: pointer to variables of adc_handle_t . The variable contains the configuration information of a specified ADC. |
| Return Value | HAL status |
| Remarks | |

2.10.4.14 hal_adc_resume_reg

Table 2-68 hal_adc_resume_reg API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_adc_resume_reg(adc_handle_t *p_adc); |
| Function Description | Resume registers related to ADC configuration during wakeup. |
| Parameter | p_adc: pointer to variables of adc_handle_t . The variable contains the configuration information of a specified ADC. |

| | |
|--------------|------------|
| Return Value | HAL status |
| Remarks | |

2.11 HAL DMA Generic Driver

2.11.1 DMA Driver Functionalities

The HAL DMA driver features the following functionalities:

- Two operation modes: normal and circular
- Four transfer directions: peripheral to memory, memory to peripheral, peripheral to peripheral, and memory to memory
- Three address increment modes: increment, decrement, and no change
- Three data widths: byte, halfword, and word
- Configurable channel priorities
- Two data transfer modes: polling and interrupt
- Transfer complete, block transfer complete, and abort complete interrupt callback functions
- Getting operating state and error code of DMA driver

2.11.2 How to Use DMA Driver

Developers can use the DMA driver in the following scenarios:

1. Enable and configure peripherals to be connected to DMA channels (except for SRAM memories: no initialization is required).
2. For a given channel, use `hal_dma_init()` to configure the parameters: DMA source/destination peripheral, transfer direction, source/destination data format, circular/normal mode, channel priority level, and source/destination address increment mode.
3. Retrieve the DMA state through `hal_dma_get_state()`, and retrieve the DMA error code through `hal_dma_get_error()` in error detection.
4. Abort the current transfer by using `hal_dma_abort()`.

The polling mode differs from the interrupt mode in the method to judge whether transfer is completed (the polling mode requires loop detection of the completion status; the interrupt mode requires transfer complete interrupts).

Details are as below:

I/O operation in polling mode

1. Call `hal_dma_start()` to start DMA transfer after configuring the source and destination addresses as well as the length of data to be transferred.
2. Call `hal_dma_poll_for_transfer()` to poll for the DMA transfer status till the transfer completes or transfer timeout occurs. In this case, developers can set a timeout based on application requirements.

I/O operation in interrupt mode

1. Configure the DMA interrupt priority by calling `hal_nvic_set_priority()`.
2. Enable DMA interrupt handling by calling `hal_nvic_enable_irq()`.
3. Call `hal_dma_start_it()` to start DMA transfer with interrupt enabled after configuring the source and destination addresses as well as the length of data to be transferred.
4. Execute `hal_dma_irq_handler()` at the end of data transfer, and call the callback function which developers register through `hal_dma_register_callback()`.

2.11.3 DMA Driver Structures

2.11.3.1 dma_init_t

The initialization structure `dma_init_t` of DMA driver is defined below:

Table 2-69 `dma_init_t` structure

| Data Field | Field Description | Value |
|-----------------------------------|------------------------|---|
| <code>uint32_t src_request</code> | Source peripheral type | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • <code>DMA_REQUEST_SPIM_TX</code> (SPIM TX) • <code>DMA_REQUEST_SPIM_RX</code> (SPIM RX) • <code>DMA_REQUEST_SPIS_TX</code> (SPIS TX) • <code>DMA_REQUEST_SPIS_RX</code> (SPIS RX) • <code>DMA_REQUEST_QSPI0_TX</code> (QSPI0 TX) • <code>DMA_REQUEST_QSPI0_RX</code> (QSPI0 RX) • <code>DMA_REQUEST_I2C0_TX</code> (I2C0 TX) • <code>DMA_REQUEST_I2C0_RX</code> (I2C0 RX) • <code>DMA_REQUEST_I2C1_TX</code> (I2C1 TX) • <code>DMA_REQUEST_I2C1_RX</code> (I2C1 RX) • <code>DMA_REQUEST_I2S_S_TX</code> (I2SS TX) • <code>DMA_REQUEST_I2S_S_RX</code> (I2SS RX) • <code>DMA_REQUEST_UART0_TX</code> (UART0 TX) • <code>DMA_REQUEST_UART0_RX</code> (UART0 RX) • <code>DMA_REQUEST_QSPI1_TX</code> (QSPI1 TX) • <code>DMA_REQUEST_QSPI1_RX</code> (QSPI1 RX) • <code>DMA_REQUEST_I2S_M_TX</code> (I2SM TX) • <code>DMA_REQUEST_I2S_M_RX</code> (I2SM RX) • <code>DMA_REQUEST_SNSADC</code> (Sense ADC) |

| Data Field | Field Description | Value |
|------------------------|--|--|
| | | <ul style="list-style-type: none"> DMA_REQUEST_MEM (memory) |
| uint32_t dst_request | Destination peripheral type | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> DMA_REQUEST_SPIM_TX (SPIM TX) DMA_REQUEST_SPIM_RX (SPIM RX) DMA_REQUEST_SPIS_TX (SPIS TX) DMA_REQUEST_SPIS_RX (SPIS RX) DMA_REQUEST_QSPI0_TX (QSPI0 TX) DMA_REQUEST_QSPI0_RX (QSPI0 RX) DMA_REQUEST_I2CO_TX (I2CO TX) DMA_REQUEST_I2CO_RX (I2CO RX) DMA_REQUEST_I2C1_TX (I2C1 TX) DMA_REQUEST_I2C1_RX (I2C1 RX) DMA_REQUEST_I2S_S_TX (I2SS TX) DMA_REQUEST_I2S_S_RX (I2SS RX) DMA_REQUEST_UART0_TX (UART0 TX) DMA_REQUEST_UART0_RX (UART0 RX) DMA_REQUEST_QSPI1_TX (QSPI1 TX) DMA_REQUEST_QSPI1_RX (QSPI1 RX) DMA_REQUEST_I2S_M_TX (I2SM TX) DMA_REQUEST_I2S_M_RX (I2SM RX) DMA_REQUEST_SNSADC (Sense ADC) DMA_REQUEST_MEM (memory) |
| uint32_t direction | Data transfer direction | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> DMA_MEMORY_TO_MEMORY (memory to memory) DMA_MEMORY_TO_PERIPH (memory to peripheral) DMA_PERIPH_TO_MEMORY (peripheral to memory) DMA_PERIPH_TO_PERIPH (peripheral to peripheral) |
| uint32_t src_increment | Increment mode for source address | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> DMA_SRC_INCREMENT (source address increments) DMA_SRC_DECREMENT (source address decrements) DMA_SRC_NO_CHANGE (source address remains unchanged) |
| uint32_t dst_increment | Increment mode for destination address | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> DMA_DST_INCREMENT (destination address increments) |

| Data Field | Field Description | Value |
|-----------------------------|---|---|
| | | <ul style="list-style-type: none"> DMA_DST_DECREMENT (destination address decrements) DMA_DST_NO_CHANGE (destination address remains unchanged) |
| uint32_t src_data_alignment | Data width/alignment of source address | This parameter can be one of the following values: <ul style="list-style-type: none"> DMA_SDATAALIGN_BYTE (byte-aligned) DMA_SDATAALIGN_HALFWORD (halfword-aligned) DMA_SDATAALIGN_WORD (word-aligned) |
| uint32_t dst_data_alignment | Data width/alignment of destination address | This parameter can be one of the following values: <ul style="list-style-type: none"> DMA_DDATAALIGN_BYTE (byte-aligned) DMA_DDATAALIGN_HALFWORD (halfword-aligned) DMA_DDATAALIGN_WORD (word-aligned) |
| uint32_t mode | Operation mode | This parameter can be one of the following values: <ul style="list-style-type: none"> DMA_NORMAL (normal mode; single transfer) DMA_CIRCULAR (circular mode; multiple transfers) |
| uint32_t priority | Channel priority level | This parameter can be one of the following values: <ul style="list-style-type: none"> DMA_PRIORITY_LOW (low priority level) DMA_PRIORITY_MEDIUM (medium priority level) DMA_PRIORITY_HIGH (high priority level) DMA_PRIORITY_VERY_HIGH (very high priority level) |

2.11.3.2 dma_handle_t

The dma_handle_t structure of DMA driver is defined below:

Table 2-70 dma_handle_t structure

| Data Field | Field Description | Value |
|-----------------------|----------------------|---|
| dma_channel_t channel | DMA channel instance | This parameter can be one of the following values: <ul style="list-style-type: none"> DMA_Channel0 (Channel 0) DMA_Channel1 (Channel 1) DMA_Channel2 (Channel 2) DMA_Channel3 (Channel 3) DMA_Channel4 (Channel 4) DMA_Channel5 (Channel 5) DMA_Channel6 (Channel 6) |

| Data Field | Field Description | Value |
|--|--|---|
| | | <ul style="list-style-type: none"> DMA_Channel7 (Channel 7) |
| <code>dma_init_t init</code> | Initialization structure | See " Section 2.11.3.1 dma_init_t ". |
| <code>__IO hal_lock_t lock</code> | DMA lock (initialization by developers not required) | N/A |
| <code>__IO hal_dma_state_t state</code> | DMA operating state (initialization by developers not required) | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> HAL_DMA_STATE_RESET (not initialized) HAL_DMA_STATE_READY (initialized and ready for use) HAL_DMA_STATE_BUSY (busy) HAL_DMA_STATE_TIMEOUT (timeout) HAL_DMA_STATE_ERROR (error) |
| <code>void *p_parent</code> | Pointer of the peripheral handle that uses the current channel instance can be the handle pointer of other peripherals. | N/A |
| <code>void (*xfer_tfr_callback)(struct __dma_handle_t *p_dma)</code> | Transfer complete callback function. This function can be registered and unregistered by using <code>hal_dma_register_callback()</code> and <code>hal_dma_unregister_callback()</code> . | N/A |
| <code>void (*xfer_blk_callback)(struct __dma_handle_t *p_dma)</code> | Block transfer complete callback function. This function can be registered and unregistered by using <code>hal_dma_register_callback()</code> and <code>hal_dma_unregister_callback()</code> . | N/A |
| <code>void (*xfer_error_callback)(struct __dma_handle_t *p_dma)</code> | Error callback function. This function can be registered and unregistered by using <code>hal_dma_register_callback()</code> and <code>hal_dma_unregister_callback()</code> . | N/A |
| <code>void (*xfer_abort_callback)(struct __dma_handle_t *p_dma)</code> | Abort complete callback function This function can be registered and unregistered by using <code>hal_dma_register_callback()</code> and <code>hal_dma_unregister_callback()</code> . | N/A |
| <code>__IO uint32_t error_code</code> | DMA error code (initialization by developers not required) | This parameter can be one of the following values: |

| Data Field | Field Description | Value |
|------------------------|---|--|
| | | <ul style="list-style-type: none"> • HAL_DMA_ERROR_NONE (no error) • HAL_DMA_ERROR_TE (transfer error) • HAL_DMA_ERROR_NO_XFER (no ongoing transfer) • HAL_DMA_ERROR_TIMEOUT (timeout) |
| uint32_t retention[5]; | DMA register information (managed by DMA drivers and initialization by developers not required) | N/A |

2.11.4 DMA Driver APIs

The DMA driver APIs are listed in the table below:

Table 2-71 DMA driver APIs

| API Type | API Name | Description |
|---------------------------------|-------------------------------|---|
| Initialization | hal_dma_init() | Initialize a specified DMA channel, and configure the parameters of the peripheral in DMA transfer and destination peripherals. |
| | hal_dma_deinit() | Deinitialize a specified DMA channel. |
| I/O operation | hal_dma_start() | Start DMA transfer in polling mode. No interrupt occurs at the end of the transfer. |
| | hal_dma_start_it() | Start DMA transfer in interrupt mode. An interrupt occurs at the end of the transfer. |
| | hal_dma_poll_for_transfer() | Poll for DMA transfer. The function is considered successful when the transfer ends. It should be used in association with hal_dma_start(). |
| | hal_dma_abort() | Abort DMA transfer. Abort complete callback function will not be called when the abortion completes. |
| | hal_dma_abort_it() | Abort DMA transfer. Abort complete callback function will be called when the abortion completes. |
| Interrupt handling and callback | hal_dma_irq_handler() | Interrupt handler |
| | hal_dma_register_callback() | Register DMA interrupt callbacks. |
| | hal_dma_unregister_callback() | Unregister DMA interrupt callbacks. |
| State and error | hal_dma_get_state() | Get the driver operating state. |
| | hal_dma_get_error() | Get error code. |
| Sleep | hal_dma_suspend_reg() | Suspend registers related to DMA configuration in sleep mode. |
| | hal_dma_resume_reg() | Resume registers related to DMA configuration during wakeup. |

The sections below elaborate on these APIs.

2.11.4.1 hal_dma_init

Table 2-72 hal_dma_init API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_dma_init(dma_handle_t *p_dma) |
| Function Description | Initialize a specified DMA channel according to the specified parameters in dma_init_t. |
| Parameter | p_dma: pointer to variables of dma_handle_t. The variable contains the configuration information of a specified DMA channel. |
| Return Value | HAL status |
| Remarks | This function only initializes DMA channels. Call hal_dma_start() and hal_dma_start_it() to enable DMA transfer. |

2.11.4.2 hal_dma_deinit

Table 2-73 hal_dma_deinit API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_dma_deinit(dma_handle_t *p_dma) |
| Function Description | Deinitialize DMA peripherals. |
| Parameter | p_dma: pointer to variables of dma_handle_t. The variable contains the configuration information of a specified DMA channel. |
| Return Value | HAL status |
| Remarks | |

2.11.4.3 hal_dma_start

Table 2-74 hal_dma_start API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_dma_start(dma_handle_t *p_dma, uint32_t src_address, uint32_t dst_address, uint32_t data_length) |
| Function Description | Start DMA transfer. |
| Parameter | <p>p_dma: pointer to variables of dma_handle_t. The variable contains the configuration information of a specified DMA channel.</p> <p>src_address: the source memory buffer address. This parameter shall be aligned with src_data_alignment set in hal_dma_init().</p> <p>dst_address: the destination memory buffer address. This parameter shall be aligned with dst_data_alignment set in hal_dma_init().</p> <p>data_length: the length of data to be transferred from the source to the destination. The minimum data unit of this parameter shall be the same as that in src_data_alignment.</p> |
| Return Value | HAL status |

| | |
|---------|--|
| Remarks | |
|---------|--|

2.11.4.4 hal_dma_start_it

Table 2-75 hal_dma_start_it API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_dma_start_it(dma_handle_t *p_dma, uint32_t src_address, uint32_t dst_address, uint32_t data_length) |
| Function Description | Start DMA transfer with interrupt enabled. |
| Parameter | <p>p_dma: pointer to variables of dma_handle_t. The variable contains the configuration information of a specified DMA channel.</p> <p>src_address: the source memory buffer address. This parameter shall be aligned with src_data_alignment set in hal_dma_init().</p> <p>dst_address: the destination memory buffer address. This parameter shall be aligned with dst_data_alignment set in hal_dma_init().</p> <p>data_length: the length of data to be transferred from the source to the destination. This parameter shall be aligned with src_data_alignment set in hal_dma_init().</p> |
| Return Value | HAL status |
| Remarks | |

2.11.4.5 hal_dma_abort

Table 2-76 hal_dma_abort API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_dma_abort(dma_handle_t *p_dma) |
| Function Description | Abort DMA transfer. |
| Parameter | p_dma: pointer to variables of dma_handle_t . The variable contains the configuration information of a specified DMA channel. |
| Return Value | HAL status |
| Remarks | |

2.11.4.6 hal_dma_abort_it

Table 2-77 hal_dma_abort_it API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_dma_abort_it(dma_handle_t *p_dma) |
| Function Description | Abort DMA transfer in interrupt mode. |
| Parameter | p_dma: pointer to variables of dma_handle_t . The variable contains the configuration information of a specified DMA channel. |
| Return Value | HAL status |

| | |
|---------|--|
| Remarks | |
|---------|--|

2.11.4.7 hal_dma_poll_for_transfer

Table 2-78 hal_dma_poll_for_transfer API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_dma_poll_for_transfer(dma_handle_t *p_dma, uint32_t timeout) |
| Function Description | Poll for transfer complete. |
| Parameter | p_dma: pointer to variables of dma_handle_t . The variable contains the configuration information of a specified DMA channel. timeout: timeout period |
| Return Value | HAL status |
| Remarks | |

2.11.4.8 hal_dma_irq_handler

Table 2-79 hal_dma_irq_handler API

| | |
|-----------------------------|---|
| Function Prototype | void hal_dma_irq_handler(dma_handle_t *p_dma) |
| Function Description | Handle DMA interrupt requests. |
| Parameter | p_dma: pointer to variables of dma_handle_t . The variable contains the configuration information of a specified DMA channel. |
| Return Value | None |
| Remarks | |

2.11.4.9 hal_dma_register_callback

Table 2-80 hal_dma_register_callback API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_dma_register_callback(dma_handle_t *p_dma, hal_dma_callback_id_t id, void (*callback)(dma_handle_t *p_dma)) |
| Function Description | Register callbacks. |
| Parameter | p_dma: pointer to variables of dma_handle_t . The variable contains the configuration information of a specified DMA channel. id: callback type ID. This parameter can be one of the following values: <ul style="list-style-type: none"> • HAL_DMA_XFER_TFR_CB_ID (transfer complete callback function ID) • HAL_DMA_XFER_BLK_CB_ID (block transfer complete callback function ID) • HAL_DMA_XFER_ABORT_CB_ID (abort complete callback function ID) callback: pointer to private callback function |

| | |
|---------------------|------------|
| Return Value | HAL status |
| Remarks | |

2.11.4.10 hal_dma_unregister_callback

Table 2-81 hal_dma_unregister_callback API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_dma_unregister_callback(dma_handle_t *p_dma, hal_dma_callback_id_t callback_id) |
| Function Description | Unregister callbacks. |
| Parameter | <p>p_dma: pointer to variables of dma_handle_t. The variable contains the configuration information of a specified DMA channel.</p> <p>callback_id: callback type ID. This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • HAL_DMA_XFER_TFR_CB_ID (transfer complete callback function ID) • HAL_DMA_XFER_BLK_CB_ID (block transfer complete callback function ID) • HAL_DMA_XFER_ABORT_CB_ID (abort complete callback function ID) • HAL_DMA_XFER_ALL_CB_ID (IDs for all complete callback functions) |
| Return Value | HAL status |
| Remarks | |

2.11.4.11 hal_dma_get_state

Table 2-82 hal_dma_get_state API

| | |
|-----------------------------|--|
| Function Prototype | hal_dma_state_t hal_dma_get_state(dma_handle_t *p_dma) |
| Function Description | Get DMA operating state. |
| Parameter | p_dma: pointer to variables of dma_handle_t . The variable contains the configuration information of a specified DMA channel. |
| Return Value | <p>The DMA operating state. This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • HAL_DMA_STATE_RESET (not initialized) • HAL_DMA_STATE_READY (initialized and ready for use) • HAL_DMA_STATE_BUSY (busy) • HAL_DMA_STATE_TIMEOUT (timeout) • HAL_DMA_STATE_ERROR (error) |
| Remarks | |

2.11.4.12 hal_dma_get_error

Table 2-83 hal_dma_get_error API

| | |
|-----------------------------|---|
| Function Prototype | hal_dma_state_t hal_dma_get_state(dma_handle_t *p_dma) |
| Function Description | Get the DMA error code. |
| Parameter | p_dma: pointer to variables of dma_handle_t . The variable contains the configuration information of a specified DMA channel. |
| Return Value | DMA error code. This parameter can be one of the following values: <ul style="list-style-type: none"> • HAL_DMA_ERROR_NONE (no error) • HAL_DMA_ERROR_TE (transfer error) • HAL_DMA_ERROR_NO_XFER (no ongoing transfer) • HAL_DMA_ERROR_TIMEOUT (timeout) |
| Remarks | |

2.11.4.13 hal_dma_suspend_reg

Table 2-84 hal_dma_suspend_reg API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_dma_suspend_reg(dma_handle_t *p_dma) |
| Function Description | Suspend registers related to DMA configuration in sleep mode. |
| Parameter | p_dma: pointer to variables of dma_handle_t . The variable contains the configuration information of a specified DMA channel. |
| Return Value | HAL status |
| Remarks | |

2.11.4.14 hal_dma_resume_reg

Table 2-85 hal_dma_resume_reg API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_dma_resume_reg(dma_handle_t *p_dma); |
| Function Description | Resume registers related to DMA configuration during wakeup. |
| Parameter | p_dma: pointer to variables of dma_handle_t . The variable contains the configuration information of a specified DMA channel. |
| Return Value | HAL status |
| Remarks | |

2.12 HAL DUAL TIMER Generic Driver

2.12.1 DUAL TIMER Driver Functionalities

The HAL DUAL TIMER driver features the following functionalities:

- Programmable 32-bit initial counting values
- Three clock dividers: divided by 1, divided by 16, and divided by 256
- Two counting modes: one-pulse mode and cyclic mode
- Two counting approaches: polling and interrupt
- Stopping counting in polling/interrupt mode
- Counting complete interrupt callback function
- Getting the operating state of HAL DUAL TIMER driver

2.12.2 How to Use DUAL TIMER Driver

Developers can use the DUAL TIMER driver in the following scenarios:

1. Declare a `dual_timer_handle_t` handle structure, for example: `dual_timer_handle_t dtim_handle`.
2. Initialize the DUAL TIMER low-level resources by overwriting `hal_dual_timer_base_msp_init()`. If `hal_dual_timer_base_start_it()` is used to count, developers need to call related NVIC APIs:
 - `hal_nvic_set_priority()` to configure the DUAL TIMER interrupt priority.
 - `hal_nvic_enable_irq()` to enable handling of DUAL TIMER interrupts.
3. Configure the initial counting value, counting mode, and clock divider in the init structure of `dtim_handle`.
4. Initialize the DUAL TIMER peripheral by calling `hal_dual_timer_base_init()` API.
5. If you count by executing `hal_dual_timer_base_start()` API in polling mode, you can call `hal_dual_timer_get_state()` to retrieve the operating state of the driver, so as to check whether the current counting completes.
6. If you count by executing `hal_dual_timer_base_start_it()` in interrupt mode, you can overwrite the interrupt callback `hal_dual_timer_period_elapsed_callback()`. When the DUAL TIMER completes counting and interrupt is triggered, the callback function is called automatically.
7. If one-pulse mode is used, the DUAL TIMER stops when the counting completes. You need to re-initialize the DUAL TIMER to start the timer for a next count. If cyclic mode is used, the DUAL TIMER reloads the initial counting value at the end of a count to start a next count.

2.12.3 DUAL TIMER Driver Structures

2.12.3.1 dual_timer_init_t

The initialization structure `dual_timer_init_t` of the DUAL TIMER driver is defined below:

Table 2-86 `dual_timer_init_t` structure

| Data Field | Field Description | Value |
|---------------------------------|-------------------|--|
| <code>uint32_t prescaler</code> | Frequency divider | This parameter can be one of the following values: |

| Data Field | Field Description | Value |
|-----------------------|------------------------------------|---|
| | | <ul style="list-style-type: none"> DUAL_TIMER_PRESCALER_DIV0 (fclk) DUAL_TIMER_PRESCALER_DIV16 (fclk/16) DUAL_TIMER_PRESCALER_DIV256 (fclk/256) |
| uint32_t counter_mode | Counting mode | This parameter can be one of the following values: <ul style="list-style-type: none"> DUAL_TIMER_COUNTERMODE_LOOP (cyclic mode) DUAL_TIMER_COUNTERMODE_ONESHOT (one-pulse mode) |
| uint32_t auto_reload | Automatically loaded counter value | Range: 0x0000_0000 to 0xFFFF_FFFF |

2.12.3.2 dual_timer_handle_t

The dual_timer_handle_t structure of DUAL TIMER driver is defined below:

Table 2-87 dual_timer_handle_t structure

| Data Field | Field Description | Value |
|-----------------------------------|--|---|
| dual_timer_regs_t *p_instance | TIMER peripheral instance | This parameter can be one of the following values: <ul style="list-style-type: none"> DUAL_TIMER0 DUAL_TIMER1 |
| dual_timer_init_t init | Initialization structure | See " Section 2.12.3.1 dual_timer_init_t ". |
| __IO hal_lock_t lock | DUAL TIMER lock (initialization by developers not required) | N/A |
| __IO hal_dual_timer_state_t state | DUAL TIMER operating state (initialization by developers not required) | This parameter can be one of the following values: <ul style="list-style-type: none"> HAL_DUAL_TIMER_STATE_RESET (not initialized) HAL_DUAL_TIMER_STATE_READY (initialized and ready for use) HAL_DUAL_TIMER_STATE_BUSY (busy) HAL_DUAL_TIMER_STATE_ERROR (error) |

2.12.4 DUAL TIMER Driver APIs

The DUAL TIMER driver APIs are listed in the table below:

Table 2-88 DUAL TIMER driver APIs

| API Type | API Name | Description |
|---------------------------------|--|--|
| Initialization | hal_dual_timer_base_init() | Initialize DUAL TIMER peripherals, and configure initial counting values and other parameters. |
| | hal_dual_timer_base_deinit() | Deinitialize DUAL TIMER peripherals. |
| | hal_dual_timer_base_msp_init() | Initialize NVIC interrupts used by DUAL TIMER peripherals. |
| | hal_dual_timer_base_msp_deinit() | Deinitialize NVIC interrupts used by DUAL TIMER peripherals. |
| I/O operation | hal_dual_timer_base_start() | Start counting in polling mode. |
| | hal_dual_timer_base_stop() | Stop counting in polling mode. |
| | hal_dual_timer_base_start_it() | Start counting in interrupt mode. |
| | hal_dual_timer_base_stop_it() | Stop counting in interrupt mode. |
| Control | hal_dual_timer_set_config() | Configure the timer. |
| Interrupt handling and callback | hal_dual_timer_irq_handler() | Interrupt handler |
| | hal_dual_timer_period_elapsed_callback() | Interrupt callback function at the end of counting |
| State and error | hal_dual_timer_get_state() | Get the driver operating state. |

The sections below elaborate on these APIs.

2.12.4.1 hal_dual_timer_base_init

Table 2-89 hal_dual_timer_base_init API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_dual_timer_base_init(dual_timer_handle_t *p_dual_timer) |
| Function Description | Initialize the DUAL TIMER time base unit and relevant handles according to parameters specified in dual_timer_handle_t . |
| Parameter | p_dual_timer: pointer to variables of dual_timer_handle_t . The variable contains the configuration information of a specified DUAL TIMER. |
| Return Value | HAL status |
| Remarks | |

2.12.4.2 hal_dual_timer_base_deinit

Table 2-90 hal_dual_timer_base_deinit API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_dual_timer_base_deinit(dual_timer_handle_t *p_dual_timer) |
| Function Description | Deinitialize DUAL TIMER peripherals. |

| | |
|---------------------|--|
| Parameter | p_dual_timer: pointer to variables of dual_timer_handle_t . The variable contains the configuration information of a specified DUAL TIMER. |
| Return Value | HAL status |
| Remarks | |

2.12.4.3 hal_dual_timer_base_msp_init

Table 2-91 hal_dual_timer_base_msp_init API

| | |
|-----------------------------|--|
| Function Prototype | void hal_dual_timer_base_msp_init(dual_timer_handle_t *p_dual_timer) |
| Function Description | Initialize NVIC interrupts used by DUAL TIMER peripherals. |
| Parameter | p_dual_timer: pointer to variables of dual_timer_handle_t . The variable contains the configuration information of a specified DUAL TIMER. |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite the API to initialize NVIC interrupts. |

2.12.4.4 hal_dual_timer_base_msp_deinit

Table 2-92 hal_dual_timer_base_msp_deinit API

| | |
|-----------------------------|--|
| Function Prototype | void hal_dual_timer_msp_base_deinit(dual_timer_handle_t *p_dual_timer) |
| Function Description | Deinitialize NVIC interrupts used by DUAL TIMER peripherals. |
| Parameter | p_dual_timer: pointer to variables of dual_timer_handle_t . The variable contains the configuration information of a specified DUAL TIMER. |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite the API to deinitialize NVIC interrupts. |

2.12.4.5 hal_dual_timer_base_start

Table 2-93 hal_dual_timer_base_start API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_dual_timer_base_start(dual_timer_handle_t *p_dual_timer) |
| Function Description | Enable DUAL TIMER peripherals, and start counting in polling mode. |
| Parameter | p_dual_timer: pointer to variables of dual_timer_handle_t . The variable contains the configuration information of a specified DUAL TIMER. |
| Return Value | HAL status |

| | |
|----------------|--|
| Remarks | The API does not enable DUAL TIMER interrupts. Developers are required to call hal_dual_timer_get_state() to get the counting state. |
|----------------|--|

2.12.4.6 hal_dual_timer_base_stop

Table 2-94 hal_dual_timer_base_stop API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_dual_timer_base_stop(dual_timer_handle_t *p_dual_timer) |
| Function Description | Disable DUAL TIMER peripherals, and stop counting in polling mode. |
| Parameter | p_dual_timer: pointer to variables of dual_timer_handle_t . The variable contains the configuration information of a specified DUAL TIMER. |
| Return Value | HAL status |
| Remarks | The API does not disable DUAL TIMER interrupts. Developers can execute hal_dual_timer_base_start() when calling the API. |

2.12.4.7 hal_dual_timer_base_start_it

Table 2-95 hal_dual_timer_base_start_it API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_dual_timer_base_start_it(dual_timer_handle_t *p_dual_timer) |
| Function Description | Enable DUAL TIMER peripherals, and start counting in interrupt mode. |
| Parameter | p_dual_timer: pointer to variables of dual_timer_handle_t . The variable contains the configuration information of a specified DUAL TIMER. |
| Return Value | HAL status |
| Remarks | The API enables DUAL TIMER interrupts. It returns to hal_dual_timer_period_elapsed_callback() when counting completes. |

2.12.4.8 hal_dual_timer_base_stop_it

Table 2-96 hal_dual_timer_base_stop_it API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_dual_timer_base_stop_it(dual_timer_handle_t *p_dual_timer) |
| Function Description | Disable DUAL TIMER peripherals, and stop counting in interrupt mode. |
| Parameter | p_dual_timer: pointer to variables of dual_timer_handle_t . The variable contains the configuration information of a specified DUAL TIMER. |
| Return Value | HAL status |
| Remarks | The API disables DUAL TIMER interrupts. Developers can execute hal_dual_timer_base_start_it() when calling the API. |

2.12.4.9 hal_dual_timer_set_config

Table 2-97 hal_dual_timer_set_config API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_dual_timer_set_config(dual_timer_handle_t *p_dual_timer, dual_timer_init_t *p_structure) |
| Function Description | Configure the DUAL TIMER. |
| Parameter | p_dual_timer: pointer to variables of dual_timer_handle_t . The variable contains the configuration information of a specified DUAL TIMER. p_structure: pointer to variables of dual_timer_handle_t . The variable contains the configuration information of a specified DUAL TIMER counter. |
| Return Value | HAL status |
| Remarks | |

2.12.4.10 hal_dual_timer_irq_handler

Table 2-98 hal_dual_timer_irq_handler API

| | |
|-----------------------------|--|
| Function Prototype | void hal_dual_timer_irq_handler(dual_timer_handle_t *p_dual_timer) |
| Function Description | Handle DUAL TIMER interrupts. |
| Parameter | p_dual_timer: pointer to variables of dual_timer_handle_t . The variable contains the configuration information of a specified DUAL TIMER. |
| Return Value | None |
| Remarks | The API is preferably called to reconfigure the timer after initializing DUAL TIMER. |

2.12.4.11 hal_dual_timer_period_elapsed_callback

Table 2-99 hal_dual_timer_period_elapsed_callback API

| | |
|-----------------------------|--|
| Function Prototype | void hal_dual_timer_period_elapsed_callback(dual_timer_handle_t *p_dual_timer) |
| Function Description | Interrupt callback function at the end of counting for DUAL TIMER |
| Parameter | p_dual_timer: pointer to variables of dual_timer_handle_t . The variable contains the configuration information of a specified DUAL TIMER. |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite this function before using it. |

2.12.4.12 hal_dual_timer_get_state

Table 2-100 hal_dual_timer_get_state API

| | |
|-----------------------------|--|
| Function Prototype | hal_dual_timer_state_t hal_dual_timer_get_state(dual_timer_handle_t *p_dual_timer) |
| Function Description | Return the DUAL TIMER handle status. |

| | |
|---------------------|---|
| Parameter | p_dual_timer: pointer to variables of dual_timer_handle_t . The variable contains the configuration information of a specified DUAL TIMER. |
| Return Value | The DUAL TIMER state. This parameter can be one of the following values: <ul style="list-style-type: none"> • HAL_DUAL_TIMER_STATE_RESET (not initialized) • HAL_DUAL_TIMER_STATE_READY (initialized and ready for use) • HAL_DUAL_TIMER_STATE_BUSY (busy) • HAL_DUAL_TIMER_STATE_ERROR (error) |
| Remarks | |

2.13 HAL AES Generic Driver

2.13.1 AES Driver Functionalities

The HAL Advanced Encryption Standard (AES) driver features the following functionalities:

- 128-bits, 192-bit, and 256-bit keys
- Encryption and decryption in Electronic Codebook (ECB) and Cipher Block Chaining (CBC) modes
- Three key loading modes: MCU, DMA, and KPORT.
- Anti-differential power analysis (DPA) attacks
- Three operating modes: polling, interrupt, and DMA
- Callback functions in interrupt and DMA modes
- Getting operating state and error code of AES driver
- Timeout settings

2.13.2 How to Use AES Driver

2.13.2.1 Initialization

To initialize the AES driver, developers can:

1. Declare an `aes_handle_t` handle structure variable, for example: `aes_handle_t aes_handle`.
2. Initialize the AES low-level resources by overwriting `hal_aes_msp_init()`. To use the interrupt mode, call related NVIC APIs:
 - Configure the AES interrupt priority by calling `hal_nvic_set_priority()`.
 - Enable AES interrupt by calling `hal_nvic_enable_irq()`.
3. Configure parameters for `p_instance` and `init` structures of `aes_handle`, including AES peripheral instance, key length, operating mode for encryption and decryption blocks, key pointer, initialization vector in CBC mode, security mode, and random number seed.

4. Initialize the AES registers by calling `hal_aes_init()`.

2.13.2.2 Encryption and Decryption in ECB Mode

The encryption/decryption in ECB mode supports three calculation modes: polling, interrupt, and DMA. The three modes differ in how to load data for calculation and how to determine the calculation is completed: The polling mode requires loop detection of the completion status; the interrupt mode and the DMA mode require calculation complete interrupts. Follow the steps below to use any one of the three modes:

I/O operation in polling mode

1. Encrypt data by using `hal_aes_ecb_encrypt()`, and decrypt data by using `hal_aes_ecb_decrypt()`.
2. Follow Step 1, until the calculation is completed or an error is returned due to timeout. If an error is returned, check the error code by calling `hal_aes_get_error()`; in the case of a large amount of data to be encrypted or decrypted, repeat Step 1.

I/O operation in interrupt mode

1. Developers can execute `hal_aes_done_callback()`, `hal_aes_error_callback()`, and `hal_aes_abort_cplt_callback()` on demand.
2. Encrypt data by using `hal_aes_ecb_encrypt_it()`, and decrypt data by using `hal_aes_ecb_decrypt_it()`.
3. If the calculation is completed, `hal_aes_done_callback()` is called; if an error occurs, `hal_aes_error_callback()` is called; in the case of a large amount of data to be encrypted or decrypted, repeat Step 2.
4. To abort the calculation, call `hal_aes_abort()` and `hal_aes_abort_it()`. The `hal_aes_abort()` API only aborts the current calculation, and `hal_aes_abort_it()` calls `hal_aes_abort_cplt_callback()` after aborting the current calculation.

I/O operation in DMA mode

1. Developers can execute `hal_aes_done_callback()`, `hal_aes_error_callback()`, and `hal_aes_abort_cplt_callback()` on demand.
2. Encrypt data by using `hal_aes_ecb_encrypt_dma()`, and decrypt data by using `hal_aes_ecb_decrypt_dma()`.
3. If the calculation is completed, `hal_aes_done_callback()` is called; if an error occurs, `hal_aes_error_callback()` is called; in the case of a large amount of data to be encrypted or decrypted, repeat Step 2.
4. To abort the calculation, call `hal_aes_abort()` and `hal_aes_abort_it()`. The `hal_aes_abort()` API only aborts the current calculation, and `hal_aes_abort_it()` calls `hal_aes_abort_cplt_callback()` after aborting the current calculation.

2.13.2.3 Encryption and Decryption in CBC Mode

The encryption/decryption in CBC mode supports three calculation modes: polling, interrupt, and DMA. The three modes differ in how to load data for calculation and how to determine the calculation is completed: The polling mode requires loop detection of the completion status; the interrupt mode and the DMA mode require calculation complete interrupts. Follow the steps below to use any one of the three modes:

I/O operation in polling mode

1. Reload `p_init_vector`; encrypt data by using `hal_aes_cbc_encrypt()`, and decrypt data by using `hal_aes_cbc_decrypt()`. If the data flow is too large in size to be encrypted or decrypted for one time, data segmentation is required. For encryption of non-start data segments, `p_init_vector` represents the last 16 bytes of the data for the last calculation result; for decryption, `p_init_vector` represents the last 16 bytes of data to be decrypted last time.
2. Follow Step 1, until the calculation is completed or an error is returned due to timeout. If an error is returned, check the error code by calling `hal_aes_get_error()`; in the case of a large amount of data to be encrypted or decrypted, repeat Step 1.

I/O operation in interrupt mode

1. Developers can execute `hal_aes_done_callback()`, `hal_aes_error_callback()`, and `hal_aes_abort_cplt_callback()` on demand.
2. Reload `p_init_vector`; encrypt data by using `hal_aes_cbc_encrypt_it()`, and decrypt data by using `hal_aes_cbc_decrypt_it()`. If the data flow is too large in size to be encrypted or decrypted for one time, data segmentation is required. For encryption of non-start data segments, `p_init_vector` represents the last 16 bytes of the data for the last calculation result; for decryption, `p_init_vector` represents the last 16 bytes of data to be decrypted last time.
3. If the calculation is completed, `hal_aes_done_callback()` is called; if an error occurs, `hal_aes_error_callback()` is called; in the case of a large amount of data to be encrypted or decrypted, repeat Step 2.
4. To abort the calculation, call `hal_aes_abort()` and `hal_aes_abort_it()`. The `hal_aes_abort()` API only aborts the current calculation, and `hal_aes_abort_it()` calls `hal_aes_abort_cplt_callback()` after aborting the current calculation.

I/O operation in DMA mode

1. Developers can execute `hal_aes_done_callback()`, `hal_aes_error_callback()`, and `hal_aes_abort_cplt_callback()` on demand.
2. Reload `p_init_vector`; encrypt data by using `hal_aes_cbc_encrypt_dma()`, and decrypt data by using `hal_aes_cbc_decrypt_dma()`. If the data flow is too large in size to be encrypted or decrypted for one time, data segmentation is required. For encryption of non-start data segments, `p_init_vector` represents the last 16 bytes of the data for the last calculation result; for decryption, `p_init_vector` represents the last 16 bytes of data to be decrypted last time.
3. If the calculation is completed, `hal_aes_done_callback()` is called; if an error occurs, `hal_aes_error_callback()` is called; in the case of a large amount of data to be encrypted or decrypted, repeat Step 2.
4. To abort the calculation, call `hal_aes_abort()` and `hal_aes_abort_it()`. The `hal_aes_abort()` API only aborts the current calculation, and `hal_aes_abort_it()` calls `hal_aes_abort_cplt_callback()` after aborting the current calculation.

2.13.3 AES Driver Structures

2.13.3.1 aes_init_t

The initialization structure aes_init_t of the AES driver is defined below:

Table 2-101 aes_init_t structure

| Data Field | Field Description | Value |
|-------------------------|---|---|
| uint32_t key_size | Key size | This parameter can be one of the following values: <ul style="list-style-type: none"> AES_KEYSIZE_128BITS (128 bits) AES_KEYSIZE_192BITS (192 bits) AES_KEYSIZE_256BITS (256 bits) |
| uint32_t operation_mode | Operating mode (encryption/ decryption) | This parameter can be one of the following values: <ul style="list-style-type: none"> AES_OPERATION_MODE_ENCRYPT (encryption) AES_OPERATION_MODE_DECRYPT (decryption) |
| uint32_t chaining_mode | Encryption/ Decryption mode for data flow | This parameter can be one of the following values: <ul style="list-style-type: none"> AES_CHAININGMODE_ECB (data flow encryption/ decryption in ECB mode) AES_CHAININGMODE_CBC (data flow encryption/ decryption in CBC mode) |
| uint32_t *p_key | Key | Pointer to the key |
| uint32_t *p_init_vector | Initialization vector, valid in CBC mode | Pointer to the initialization vector |
| uint32_t dpa_mode | To enable security mode or not | This parameter can be one of the following values: <ul style="list-style-type: none"> ENABLE (enable) DISABLE (disable) |
| uint32_t *p_seed | Random number seed for security mode | Point to 16-byte array. |

2.13.3.2 aes_handle_t

The handle structure aes_handle_t of the AES driver is defined below:

Table 2-102 aes_handle_t structure

| Data Field | Field Description | Value |
|------------------------|--------------------------|--|
| aes_regs_t *p_instance | AES peripheral instance | AES |
| aes_init_t init | Initialization structure | See " Section 2.13.3.1 aes_init_t ". |

| Data Field | Field Description | Value |
|--------------------------------|---|---|
| uint32_t *p_cryp_input_buffer | Pointer to the buffer where data flow is to be encrypted or decrypted (initialization by developers not required) | N/A |
| uint32_t *p_cryp_output_buffer | Pointer to encryption/decryption result buffer (initialization by developers not required) | N/A |
| uint32_t block_size | Size of data block to be encrypted or decrypted (initialization by developers not required) | N/A |
| uint32_t block_count | Count of data block to be encrypted or decrypted (initialization by developers not required) | The value is initialized to block_size, and decreases progressively to 0 during calculation. |
| __IO hal_lock_t lock | AES lock (initialization by developers not required) | N/A |
| __IO hal_aes_state_t state | AES operating state (initialization by developers not required) | This parameter can be one of the following values: <ul style="list-style-type: none"> • HAL_AES_STATE_RESET (not initialized) • HAL_AES_STATE_READY (initialized and ready for use) • HAL_AES_STATE_BUSY (busy) • HAL_AES_STATE_ERROR (operation error) • HAL_AES_STATE_TIMEOUT (timeout) • HAL_AES_STATE_SUSPENDED (suspended) |
| __IO uint32_t error_code | AES error code (initialization by developers not required) | This parameter can be one of the following values: <ul style="list-style-type: none"> • HAL_AES_ERROR_NONE (no error) • HAL_AES_ERROR_TIMEOUT (timeout) • HAL_AES_ERROR_TRANSFER (transfer error) • HAL_AES_ERROR_INVALID_PARAM (invalid parameter) |
| uint32_t timeout | AES timeout period (initialization by developers not required) | N/A |

| Data Field | Field Description | Value |
|------------------------|--|-------|
| uint32_t retention[18] | AES register information (managed by AES driver and initialization by developers not required) | N/A |

 **Note:**

N/A indicates that value options for the parameter are not applicable.

2.13.4 AES Driver APIs

The AES driver APIs are listed in the table below:

Table 2-103 AES driver APIs

| API Type | API Name | Description |
|----------------|---------------------------|---|
| Initialization | hal_aes_init() | Initialize the AES peripheral, and configure keys and other parameters. |
| | hal_aes_deinit() | Deinitialize the AES peripheral. |
| | hal_aes_msp_init() | Initialize NVIC interrupts used by the AES peripheral. |
| | hal_aes_msp_deinit() | Deinitialize NVIC interrupts used by the AES peripheral. |
| I/O operation | hal_aes_ecb_encrypt() | Encrypt data in ECB mode (polling mode). |
| | hal_aes_ecb_decrypt() | Decrypt data in ECB mode (polling mode). |
| | hal_aes_cbc_encrypt() | Encrypt data in CBC mode (polling mode). |
| | hal_aes_cbc_decrypt() | Decrypt data in CBC mode (polling mode). |
| | hal_aes_ecb_encrypt_it() | Encrypt data in ECB mode (interrupt mode). |
| | hal_aes_ecb_decrypt_it() | Decrypt data in ECB mode (interrupt mode). |
| | hal_aes_cbc_encrypt_it() | Encrypt data in CBC mode (interrupt mode). |
| | hal_aes_cbc_decrypt_it() | Decrypt data in CBC mode (interrupt mode). |
| | hal_aes_ecb_encrypt_dma() | Encrypt data in ECB mode (DMA mode). |
| | hal_aes_ecb_decrypt_dma() | Decrypt data in ECB mode (DMA mode). |
| | hal_aes_cbc_encrypt_dma() | Encrypt data in CBC mode (DMA mode). |
| | hal_aes_cbc_decrypt_dma() | Decrypt data in CBC mode (DMA mode). |
| | hal_aes_abort() | Abort encryption/decryption in polling mode. |
| | hal_aes_abort_it() | Abort encryption/decryption in interrupt mode. |

| API Type | API Name | Description |
|---------------------------------|-------------------------------|---|
| Interrupt handling and callback | hal_aes_irq_handler() | Interrupt handler |
| | hal_aes_done_callback() | Encryption/Decryption complete interrupt callback function |
| | hal_aes_error_callback() | Error interrupt callback function |
| | hal_aes_abort_cplt_callback() | Abort complete interrupt callback function |
| State and error | hal_aes_get_state() | Get the driver operating state. |
| | hal_aes_get_error() | Get error code. |
| Control | hal_aes_set_timeout() | Set a timeout period. |
| Sleep | hal_aes_suspend_reg() | Suspend registers related to AES configuration in sleep mode. |
| | hal_aes_resume_reg() | Resume registers related to AES configuration during wakeup. |

The sections below elaborate on these APIs.

2.13.4.1 hal_aes_init

Table 2-104 hal_aes_init API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_aes_init(aes_handle_t *p_aes) |
| Function Description | Initialize the AES peripheral according to parameters of aes_init_t . |
| Parameter | p_aes: pointer to variables of aes_handle_t . The variable contains the configuration information of a specified AES. |
| Return Value | HAL status |
| Remarks | |

2.13.4.2 hal_aes_deinit

Table 2-105 hal_aes_deinit API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_aes_deinit(aes_handle_t *p_aes) |
| Function Description | Deinitialize the AES peripheral, and restore the registers in AES module to defaults. |
| Parameter | p_aes: pointer to variables of aes_handle_t . The variable contains specified register base addresses. |
| Return Value | HAL status |
| Remarks | |

2.13.4.3 hal_aes_msp_init

Table 2-106 hal_aes_msp_init API

| | |
|-----------------------------|---|
| Function Prototype | void hal_aes_msp_init(aes_handle_t *p_aes) |
| Function Description | Initialize NVIC interrupts and DMA channels used by the AES peripheral. |
| Parameter | p_aes: pointer to variables of aes_handle_t |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite the API to initialize NVIC interrupts and DMA channels. |

2.13.4.4 hal_aes_msp_deinit

Table 2-107 hal_aes_msp_deinit API

| | |
|-----------------------------|---|
| Function Prototype | void hal_aes_msp_deinit(aes_handle_t *p_aes) |
| Function Description | Deinitialize NVIC interrupts and DMA channels used by the AES peripheral. |
| Parameter | p_aes: pointer to variables of aes_handle_t |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite the API to deinitialize NVIC interrupts and DMA channels. |

2.13.4.5 hal_aes_ecb_encrypt

Table 2-108 hal_aes_ecb_encrypt API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_aes_ecb_encrypt(aes_handle_t *p_aes, uint32_t *p_plain_data, uint32_t number, uint32_t *p_cypher_data, uint32_t timeout) |
| Function Description | Encrypt data in ECB mode (polling mode). |
| Parameter | <p>p_aes: pointer to variables of aes_handle_t</p> <p>p_plain_data: pointer to the data to be encrypted</p> <p>number: the length of data to be encrypted (in byte). The maximum number is 32768, and the number shall be an integral multiple of 16.</p> <p>p_cypher_data: pointer to the memory that stores encryption results</p> <p>timeout: timeout period (ms)</p> |
| Return Value | HAL status |
| Remarks | If the data to be encrypted exceeds the maximum length, data segmentation is required for encryption. |

2.13.4.6 hal_aes_ecb_decrypt

Table 2-109 hal_aes_ecb_decrypt API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_aes_ecb_decrypt(aes_handle_t *p_aes, uint32_t *p_cypher_data, uint32_t number, uint32_t *p_plain_data, uint32_t timeout) |
| Function Description | Decrypt data in ECB mode (polling mode). |
| Parameter | <p>p_aes: pointer to variables of aes_handle_t</p> <p>p_cypher_data: pointer to the data to be decrypted</p> <p>number: the length of data to be decrypted (in byte). The maximum number is 32768, and the number shall be an integral multiple of 16.</p> <p>p_plain_data: pointer to the memory that stores decryption results</p> <p>timeout: timeout period (ms)</p> |
| Return Value | HAL status |
| Remarks | If the data to be decrypted exceeds the maximum length, data segmentation is required for decryption. |

2.13.4.7 hal_aes_cbc_encrypt

Table 2-110 hal_aes_cbc_encrypt API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_aes_cbc_encrypt(aes_handle_t *p_aes, uint32_t *p_plain_data, uint32_t number, uint32_t *p_cypher_data, uint32_t timeout) |
| Function Description | Encrypt data in CBC mode (polling mode). |
| Parameter | <p>p_aes: pointer to variables of aes_handle_t</p> <p>p_plain_data: pointer to the data to be encrypted</p> <p>number: the length of data to be encrypted (in byte). The maximum number is 32768, and the number shall be an integral multiple of 16.</p> <p>p_cypher_data: pointer to the memory that stores encryption results</p> <p>timeout: timeout period (ms)</p> |
| Return Value | HAL status |
| Remarks | If the data to be encrypted exceeds the maximum length, data segmentation is required for encryption. In addition, from the second encryption operation, the initialization vector (p_aes->init.p_init_vector) shall be reloaded as the last 16 bytes of the data for the last encryption result. |

2.13.4.8 hal_aes_cbc_decrypt

Table 2-111 hal_aes_cbc_decrypt API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_aes_cbc_decrypt(aes_handle_t *p_aes, uint32_t *p_cypher_data, uint32_t number, uint32_t *p_plain_data, uint32_t timeout) |
| Function Description | Decrypt data in CBC mode (polling mode). |

| | |
|---------------------|--|
| Parameter | <p>p_aes: pointer to variables of aes_handle_t</p> <p>p_cypher_data: pointer to the data to be decrypted</p> <p>number: the length of data to be decrypted (in byte). The maximum number is 32768, and the number shall be an integral multiple of 16.</p> <p>p_plain_data: pointer to the memory that stores decryption results</p> <p>timeout: timeout period (ms)</p> |
| Return Value | HAL status |
| Remarks | If the data to be decrypted exceeds the maximum length, data segmentation is required for decryption. In addition, from the second decryption operation, the initialization vector (p_aes->init.p_init_vector) shall be reloaded as the last 16 bytes of the data to be decrypted last time. |

2.13.4.9 hal_aes_ecb_encrypt_it

Table 2-112 hal_aes_ecb_encrypt_it API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_aes_ecb_encrypt_it(aes_handle_t *p_aes, uint32_t *p_plain_data, uint32_t number, uint32_t *p_cypher_data) |
| Function Description | Encrypt data in ECB mode (interrupt mode). |
| Parameter | <p>p_aes: pointer to variables of aes_handle_t</p> <p>p_plain_data: pointer to the data to be encrypted</p> <p>number: the length of data to be encrypted (in byte). The maximum number is 32768, and the number shall be an integral multiple of 16.</p> <p>p_cypher_data: pointer to the memory that stores encryption results</p> |
| Return Value | HAL status |
| Remarks | If the data to be encrypted exceeds the maximum length, data segmentation is required for encryption. |

2.13.4.10 hal_aes_ecb_decrypt_it

Table 2-113 hal_aes_ecb_decrypt_it API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_aes_ecb_decrypt_it(aes_handle_t *p_aes, uint32_t *p_cypher_data, uint32_t number, uint32_t *p_plain_data) |
| Function Description | Decrypt data in ECB mode (interrupt mode). |
| Parameter | <p>p_aes: pointer to variables of aes_handle_t</p> <p>p_cypher_data: pointer to the data to be decrypted</p> <p>number: the length of data to be decrypted (in byte). The maximum number is 32768, and the number shall be an integral multiple of 16.</p> <p>p_plain_data: pointer to the memory that stores decryption results</p> |

| | |
|---------------------|---|
| Return Value | HAL status |
| Remarks | If the data to be decrypted exceeds the maximum length, data segmentation is required for decryption. |

2.13.4.11 hal_aes_cbc_encrypt_it

Table 2-114 hal_aes_cbc_encrypt_it API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_aes_cbc_encrypt_it(aes_handle_t *p_aes, uint32_t *p_plain_data, uint32_t number, uint32_t *p_cypher_data) |
| Function Description | Encrypt data in CBC mode (interrupt mode). |
| Parameter | p_aes: pointer to variables of aes_handle_t p_plain_data: pointer to the data to be encrypted number: the length of data to be encrypted (in byte). The maximum number is 32768, and the number shall be an integral multiple of 16. p_cypher_data: pointer to the memory that stores encryption results |
| Return Value | HAL status |
| Remarks | If the data to be encrypted exceeds the maximum length, data segmentation is required for encryption. In addition, from the second encryption operation, the initialization vector (p_aes->init.p_init_vector) shall be reloaded as the last 16 bytes of the data for the last encryption result. |

2.13.4.12 hal_aes_cbc_decrypt_it

Table 2-115 hal_aes_cbc_decrypt_it API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_aes_cbc_decrypt_it(aes_handle_t *p_aes, uint32_t *p_cypher_data, uint32_t number, uint32_t *p_plain_data) |
| Function Description | Decrypt data in CBC mode (interrupt mode). |
| Parameter | p_aes: pointer to variables of aes_handle_t p_cypher_data: pointer to the data to be decrypted number: the length of data to be decrypted (in byte). The maximum number is 32768, and the number shall be an integral multiple of 16. p_plain_data: pointer to the memory that stores decryption results |
| Return Value | HAL status |
| Remarks | If the data to be decrypted exceeds the maximum length, data segmentation is required for decryption. In addition, from the second decryption operation, the initialization vector (p_aes->init.p_init_vector) shall be reloaded as the last 16 bytes of the data to be decrypted last time. |

2.13.4.13 hal_aes_ecb_encrypt_dma

Table 2-116 hal_aes_ecb_encrypt_dma API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_aes_ecb_encrypt_dma(aes_handle_t *p_aes, uint32_t *p_plain_data, uint32_t number, uint32_t *p_cypher_data) |
| Function Description | Encrypt data in ECB mode (DMA mode). |
| Parameter | p_aes: pointer to variables of aes_handle_t p_plain_data: pointer to the data to be encrypted number: the length of data to be encrypted (in byte). The maximum number is 32768, and the number shall be an integral multiple of 16. p_cypher_data: pointer to the memory that stores encryption results |
| Return Value | HAL status |
| Remarks | If the data to be encrypted exceeds the maximum length, data segmentation is required for encryption. |

2.13.4.14 hal_aes_ecb_decrypt_dma

Table 2-117 hal_aes_ecb_decrypt_dma API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_aes_ecb_decrypt_dma(aes_handle_t *p_aes, uint32_t *p_cypher_data, uint32_t number, uint32_t *p_plain_data) |
| Function Description | Decrypt data in ECB mode (DMA mode). |
| Parameter | p_aes: pointer to variables of aes_handle_t p_cypher_data: pointer to the data to be decrypted number: the length of data to be decrypted (in byte). The maximum number is 32768, and the number shall be an integral multiple of 16. p_plain_data: pointer to the memory that stores decryption results |
| Return Value | HAL status |
| Remarks | If the data to be decrypted exceeds the maximum length, data segmentation is required for decryption. |

2.13.4.15 hal_aes_cbc_encrypt_dma

Table 2-118 hal_aes_cbc_encrypt_dma API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_aes_cbc_encrypt_dma(aes_handle_t *p_aes, uint32_t *p_plain_data, uint32_t number, uint32_t *p_cypher_data) |
| Function Description | Encrypt data in CBC mode (DMA mode). |
| Parameter | p_aes: pointer to variables of aes_handle_t p_plain_data: pointer to the data to be encrypted |

| | |
|---------------------|---|
| | <p>number: the length of data to be encrypted (in byte). The maximum number is 32768, and the number shall be an integral multiple of 16.</p> <p>p_cypher_data: pointer to the memory that stores encryption results</p> |
| Return Value | HAL status |
| Remarks | If the data to be encrypted exceeds the maximum length, data segmentation is required for encryption. In addition, from the second encryption operation, the initialization vector (p_aes->init.p_init_vector) shall be reloaded as the last 16 bytes of the data for the last encryption result. |

2.13.4.16 hal_aes_cbc_decrypt_dma

Table 2-119 hal_aes_cbc_decrypt_dma API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_aes_cbc_decrypt_dma(aes_handle_t *p_aes, uint32_t *p_cypher_data, uint32_t number, uint32_t *p_plain_data) |
| Function Description | Decrypt data in CBC mode (DMA mode). |
| Parameter | <p>p_aes: pointer to variables of aes_handle_t</p> <p>p_cypher_data: pointer to the data to be decrypted</p> <p>number: the length of data to be decrypted (in byte). The maximum number is 32768, and the number shall be an integral multiple of 16.</p> <p>p_plain_data: pointer to the memory that stores decryption results</p> |
| Return Value | HAL status |
| Remarks | If the data to be decrypted exceeds the maximum length, data segmentation is required for decryption. In addition, from the second decryption operation, the initialization vector (p_aes->init.p_init_vector) shall be reloaded as the last 16 bytes of the data to be decrypted last time. |

2.13.4.17 hal_aes_abort

Table 2-120 hal_aes_abort API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_aes_abort(aes_handle_t *p_aes) |
| Function Description | Abort encryption/decryption in polling mode. |
| Parameter | p_aes: pointer to variables of aes_handle_t |
| Return Value | HAL status |
| Remarks | Abort encryption/decryption in non-polling mode. It is a polling API with status returned after the encryption/decryption is aborted. |

2.13.4.18 hal_aes_abort_it

Table 2-121 hal_aes_abort_it API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_aes_abort_it(aes_handle_t *p_aes) |
| Function Description | Abort encryption/decryption in interrupt mode. |
| Parameter | p_aes: pointer to variables of aes_handle_t |
| Return Value | HAL status |
| Remarks | Abort encryption/decryption in non-polling mode. It is a non-polling API with status returned immediately; after the encryption/decryption is aborted, hal_aes_abort_cplt_callback() is called. |

2.13.4.19 hal_aes_irq_handler

Table 2-122 hal_aes_irq_handler API

| | |
|-----------------------------|---|
| Function Prototype | void hal_aes_irq_handler(aes_handle_t *p_aes) |
| Function Description | Handle AES interrupt requests. |
| Parameter | p_aes: pointer to variables of aes_handle_t |
| Return Value | None |
| Remarks | |

2.13.4.20 hal_aes_done_callback

Table 2-123 hal_aes_done_callback API

| | |
|-----------------------------|---|
| Function Prototype | void hal_aes_done_callback(aes_handle_t *p_aes) |
| Function Description | AES encryption/decryption complete callback function |
| Parameter | p_aes: pointer to variables of aes_handle_t |
| Return Value | None |
| Remarks | This function is called when the encryption/decryption is completed in interrupt/DMA mode. The function is declared empty as weak function. Developers are required to overwrite this function before using it. |

2.13.4.21 hal_aes_error_callback

Table 2-124 hal_aes_error_callback API

| | |
|-----------------------------|---|
| Function Prototype | void hal_aes_error_callback(aes_handle_t *p_aes) |
| Function Description | AES encryption/decryption error callback function |
| Parameter | p_aes: pointer to variables of aes_handle_t |
| Return Value | None |

| | |
|----------------|---|
| Remarks | This function is called when an error occurs during encryption/decryption in interrupt/DMA mode. The function is declared empty as weak function. Developers are required to overwrite this function before using it. |
|----------------|---|

2.13.4.22 hal_aes_abort_cplt_callback

Table 2-125 hal_aes_abort_cplt_callback API

| | |
|-----------------------------|---|
| Function Prototype | void hal_aes_abort_cplt_callback(aes_handle_t *p_aes) |
| Function Description | AES encryption/decryption abort callback function |
| Parameter | p_aes: pointer to variables of aes_handle_t |
| Return Value | None |
| Remarks | This function is called when the encryption/decryption is aborted in interrupt/DMA mode. The function is declared empty as weak function. Developers are required to overwrite this function before using it. |

2.13.4.23 hal_aes_get_state

Table 2-126 hal_aes_get_state API

| | |
|-----------------------------|--|
| Function Prototype | hal_aes_state_t hal_aes_get_state(aes_handle_t *p_aes) |
| Function Description | Get the AES operating state. |
| Parameter | p_aes: pointer to variables of aes_handle_t |
| Return Value | The AES operating state. This parameter can be one of the following values: <ul style="list-style-type: none"> • HAL_AES_STATE_RESET (not initialized) • HAL_AES_STATE_READY (initialized and ready for use) • HAL_AES_STATE_BUSY (busy) • HAL_AES_STATE_ERROR (operation error) • HAL_AES_STATE_TIMEOUT (timeout) • HAL_AES_STATE_SUSPENDED (suspended) |
| Remarks | |

2.13.4.24 hal_aes_get_error

Table 2-127 hal_aes_get_error API

| | |
|-----------------------------|---|
| Function Prototype | uint32 hal_aes_get_error(aes_handle_t *p_aes) |
| Function Description | Return the AES error code. |
| Parameter | p_aes: pointer to variables of aes_handle_t |

| | |
|---------------------|--|
| Return Value | <p>AES error code. This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • HAL_AES_ERROR_NONE (no error) • HAL_AES_ERROR_TIMEOUT (timeout) • HAL_AES_ERROR_TRANSFER (transfer error) • HAL_AES_ERROR_INVALID_PARAM (invalid parameter) |
| Remarks | |

2.13.4.25 hal_aes_set_timeout

Table 2-128 hal_aes_set_timeout API

| | |
|-----------------------------|---|
| Function Prototype | void hal_aes_set_timeout(aes_handle_t *p_aes) |
| Function Description | Set a timeout period for AES operations. |
| Parameter | p_aes: pointer to variables of aes_handle_t |
| Return Value | None |
| Remarks | |

2.13.4.26 hal_aes_suspend_reg

Table 2-129 hal_aes_suspend_reg API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_aes_suspend_reg(aes_handle_t *p_aes) |
| Function Description | Suspend registers related to AES configuration in sleep mode. |
| Parameter | p_aes: pointer to variables of aes_handle_t |
| Return Value | HAL status |
| Remarks | |

2.13.4.27 hal_aes_resume_reg

Table 2-130 hal_aes_resume_reg API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_aes_resume_reg(aes_handle_t *p_aes) |
| Function Description | Resume registers related to AES configuration during wakeup. |
| Parameter | p_aes: pointer to variables of aes_handle_t |
| Return Value | HAL status |
| Remarks | |

2.14 HAL HMAC Generic Driver

2.14.1 HMAC Driver Functionalities

The HAL Hash-based Message Authentication Code (HMAC) driver features the following functionalities:

- Compatible with Secure Hash Algorithm 256 (SHA-256)
- A user-defined initial hash value
- Three key loading modes: MCU, DMA, and KPORT.
- Anti-differential power analysis (DPA) attacks
- Three calculation modes: polling, interrupt, and DMA
- Callback functions

2.14.2 How to Use HMAC Driver

2.14.2.1 Initialization

To initialize the HMAC driver, developers can:

1. Declare an `hmac_handle_t` handle structure variable, for example: `hmac_handle_t hmac_handle`.
2. Initialize the HMAC low-level resources by overwriting `hal_hmac_msp_init()`. To use the interrupt or DMA mode, call related NVIC APIs:
 - Configure the HMAC interrupt priority by calling `hal_nvic_set_priority()`.
 - Enable HMAC interrupt by calling `hal_nvic_enable_irq()`.
3. Configure parameters for `p_instance` and init structures of `hmac_handle`, including HMAC peripheral instance, operating mode (HMAC or SHA), key pointer, pointer to a user-defined initial hash value, and security mode.
4. Initialize the HMAC registers by calling `hal_hmac_init()`.

2.14.2.2 Calculate Message Digests by Using SHA-256

The SHA-256 algorithm supports three calculation modes: polling, interrupt, and DMA. The three modes differ in how to load data for calculation and how to determine the calculation is completed: The polling mode requires loop detection of the completion status; the interrupt mode and the DMA mode require calculation complete interrupts. Follow the steps below to use any one of the three modes:

I/O operation in polling mode

1. Developers can reload the custom initial hash value `p_user_hash` on demand. During HMAC initialization, developers need to disable interrupt and DMA mode, and select SHA mode; calculate message digests by using `hal_hmac_sha256_digest()`. If the data flow is too large in size to be calculated for one time, data segmentation is required. For calculation of non-start data segments, `p_user_hash` shall be reloaded as the results for the last calculation.
2. Follow Step 1, until the calculation is completed or an error is returned due to timeout. If an error is returned, check the error code by calling `hal_hmac_get_error()`; in the case of a large amount of data to be calculated, repeat Step 1.

I/O operation in interrupt mode

1. Developers can execute `hal_hmac_done_callback()` and `hal_hmac_error_callback()` on demand.
2. Developers can reload `p_user_hash` on demand. During HMAC initialization, developers need to enable interrupt mode, disable DMA mode, and select SHA mode; calculate message digests by using `hal_hmac_sha256_digest()`. If the data flow is too large in size to be calculated for one time, data segmentation is required. For calculation of non-start data segments, `p_user_hash` shall be reloaded as the results for the last calculation.
3. If the calculation is completed, `hal_hmac_done_callback()` is called; if an error occurs, `hal_hmac_error_callback()` is called; in the case of a large amount of data to be calculated, repeat Step 2.

I/O operation in DMA mode

1. Developers can execute `hal_hmac_done_callback()` and `hal_hmac_error_callback()` on demand.
2. Developers can reload `p_user_hash` on demand. During HMAC initialization, developers need to disable interrupt mode, enable DMA mode, and select SHA mode; calculate message digests by using `hal_hmac_sha256_digest()`. If the data flow is too large in size to be calculated for one time, data segmentation is required. For calculation of non-start data segments, `p_user_hash` shall be reloaded as the results for the last calculation.
3. If the calculation is completed, `hal_hmac_done_callback()` is called; if an error occurs, `hal_hmac_error_callback()` is called; in the case of a large amount of data to be calculated, repeat Step 2.

2.14.2.3 Calculate Message Signatures by Using HMAC

HMAC supports three calculation modes: polling, interrupt, and DMA. The three modes differ in how to load data for calculation and how to determine the calculation is completed: The polling mode requires loop detection of the completion status; the interrupt mode and the DMA mode require calculation complete interrupts. Follow the steps below to use any one of the three modes:

I/O operation in polling mode

1. Developers can reload `p_user_hash` and `p_key` on demand. During HMAC initialization, developers need to disable interrupt and DMA mode, and select HMAC mode; calculate signatures by using `hal_hmac_sha256_digest()`.
2. If the calculation is completed, `hal_hmac_done_callback()` is called; if an error occurs, `hal_hmac_error_callback()` is called; in the case of a large amount of data to be calculated, repeat Step 1.
3. Follow the steps above, until the calculation is completed or an error is returned due to timeout. If an error is returned, check the error code by calling `hal_hmac_get_error()`.

I/O operation in interrupt mode

1. Developers can execute `hal_hmac_done_callback()` and `hal_hmac_error_callback()` on demand.
2. Developers can reload `p_user_hash` and `p_key` on demand. During HMAC initialization, developers need to enable interrupt mode, disable DMA mode, and select HMAC mode; calculate signatures by using `hal_hmac_sha256_digest()`.
3. If the calculation is completed, `hal_hmac_done_callback()` is called; if an error occurs, `hal_hmac_error_callback()` is called.

I/O operation in DMA mode

1. Developers can execute `hal_hmac_done_callback()` and `hal_hmac_error_callback()` on demand.
2. Developers can reload `p_user_hash` and `p_key` on demand. During HMAC initialization, developers need to disable interrupt mode, enable DMA mode, and select HMAC mode; calculate signatures by using `hal_hmac_sha256_digest()`.
3. Each time the calculation is completed, `hal_hmac_done_callback()` is called; if an error occurs, `hal_hmac_error_callback()` is called.

2.14.3 HMAC Driver Structures

2.14.3.1 hmac_init_t

The initialization structure `hmac_init_t` of the HMAC driver is defined below:

Table 2-131 `hmac_init_t` structure

| Data Field | Field Description | Value |
|---------------------------------------|--------------------------------------|---|
| <code>uint32_t mode</code> | Specify calculation mode. | This parameter can be one of the following values: <ul style="list-style-type: none"> • <code>HMAC_MODE_SHA</code> (SHA mode) • <code>HMAC_MODE_HMAC</code> (HMAC mode) |
| <code>uint32_t *p_key</code> | Specify a key. | Pointer to the key |
| <code>uint32_t *p_user_hash</code> | Specify a custom initial hash value. | Pointer to the custom initial hash value. |
| <code>uint32_t dpa_mode</code> | Enable/Disable security mode. | This parameter can be one of the following values: <ul style="list-style-type: none"> • <code>ENABLE</code> (enable) • <code>DISABLE</code> (disable) |
| <code>uint32_t key_fetch_type</code> | Key source | This parameter can be one of the following values: <ul style="list-style-type: none"> • <code>HAL_HMAC_KEYTYPE_MCU</code> (from MCU cconfiguration) • <code>HAL_HMAC_KEYTYPE_AHB</code> (from AHB) • <code>HAL_HMAC_KEYTYPE_KRAM</code> (from KERAM) |
| <code>uint32_t enable_irq</code> | Enable/Disable interrupt mode. | This parameter can be one of the following values: <ul style="list-style-type: none"> • <code>HAL_HMAC_ENABLE_IRQ</code> (enable interrupt mode) • <code>HAL_HMAC_DISABLE_IRQ</code> (disable interrupt mode) |
| <code>uint32_t enable_dma_mode</code> | Enable/Disable DMA mode. | This parameter can be one of the following values: <ul style="list-style-type: none"> • <code>HAL_HMAC_ENABLE_DMA</code> (enable DMA mode) • <code>HAL_HMAC_DISABLE_DMA</code> (disable DMA mode) |

2.14.3.2 hmac_handle_t

The handle structure `hmac_handle_t` of the HMAC driver is defined below:

Table 2-132 `hmac_handle_t` structure

| Data Field | Field Description | Value |
|---|--|---|
| <code>hmac_regs_t *</code> <code>p_instance</code> | HMAC peripheral instance | HMAC |
| <code>hmac_init_t init</code> | Initialization structure | See " Section 2.14.3.1 hmac_init_t ". |
| <code>uint32_t * p_message</code> | Pointer to the buffer with messages to be calculated (initialization by developers not required) | N/A |
| <code>uint32_t * p_digest</code> | Pointer to calculation result buffer (initialization by developers not required) | N/A |
| <code>uint32_t block_size</code> | Size of message block to be calculated (initialization by developers not required) | This parameter ranges from 1 to 512. |
| <code>uint32_t block_count</code> | Count of message block to be calculated (initialization by developers not required) | The value is initialized to <code>block_size</code> , and decreases progressively to 0 during calculation. |
| <code>uint32_t is_last_trans</code> | Last calculation block (initialization by developers not required) | N/A |
| <code>__IO hal_lock_t lock</code> | HMAC lock (initialization by developers not required) | N/A |
| <code>__IO hal_hmac_state_t state</code> | HMAC operating state (initialization by developers not required) | This parameter can be one of the following values: <ul style="list-style-type: none"> • <code>HAL_HMAC_STATE_RESET</code> (not initialized) • <code>HAL_HMAC_STATE_READY</code> (initialized and ready for use) • <code>HAL_HMAC_STATE_BUSY</code> (busy) • <code>HAL_HMAC_STATE_ERROR</code> (error) • <code>HAL_HMAC_STATE_TIMEOUT</code> (timeout) • <code>HAL_HMAC_STATE_SUSPENDED</code> (suspended) |
| <code>__IO uint32_t error_code</code> | HMAC error code (initialization by developers not required) | This parameter can be one of the following values: <ul style="list-style-type: none"> • <code>HAL_HMAC_ERROR_NONE</code> (no error) • <code>HAL_HMAC_ERROR_TIMEOUT</code> (timeout) • <code>HAL_HMAC_ERROR_TRANSFER</code> (transfer error) • <code>HAL_HMAC_ERROR_INVALID_PARAM</code> (invalid parameter) |
| <code>uint32_t timeout</code> | HMAC calculation timeout period (initialization by developers not required) | N/A |
| <code>uint32_t retention[17]</code> | HMAC register information (managed by HMAC driver and initialization by developers not required) | N/A |

2.14.4 HMAC Driver APIs

The HMAC driver APIs are listed in the table below:

Table 2-133 HMAC driver APIs

| API Type | API Name | Description |
|---------------------------------|---------------------------|---|
| Initialization | hal_hmac_init() | Initialize the HMAC peripheral, and configure calculation mode and other parameters. |
| | hal_hmac_deinit() | Deinitialize the HMAC peripheral. |
| | hal_hmac_msp_init() | Initialize NVIC interrupts and DMA channels used by the HMAC peripheral. |
| | hal_hmac_msp_deinit() | Deinitialize NVIC interrupts and DMA channels used by the HMAC peripheral. |
| I/O operation | hal_hmac_sha256_digest() | Select polling, interrupt, or DMA mode by configuring enable_irq and enable_dma_mode. Select SHA or HMAC mode by configuring the mode. |
| Interrupt handling and callback | hal_hmac_irq_handler() | Interrupt handler |
| | hal_hmac_done_callback() | Calculation complete interrupt callback function |
| | hal_hmac_error_callback() | Error interrupt callback function |
| State and error | hal_hmac_get_state() | Get the driver operating state. |
| | hal_hmac_get_error() | Get error code. |
| Control | hal_hmac_set_timeout() | Set a timeout period. |
| Sleep | hal_hmac_suspend_reg() | Suspend registers related to HMAC configuration in sleep mode. |
| | hal_hmac_resume_reg() | Resume registers related to HMAC configuration during wakeup. |

The sections below elaborate on these APIs.

2.14.4.1 hal_hmac_init

Table 2-134 hal_hmac_init API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_hmac_init(hmac_handle_t *p_hmac) |
| Function Description | Initialize the HMAC peripheral according to parameters of hmac_init_t . |
| Parameter | p_hmac: pointer to variables of hmac_handle_t . The variable contains the configuration information of a specified HMAC. |
| Return Value | HAL status |
| Remarks | |

2.14.4.2 hal_hmac_deinit

Table 2-135 hal_hmac_deinit API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_hmac_deinit(hmac_handle_t *p_hmac) |
| Function Description | Deinitialize the HMAC peripheral. |
| Parameter | p_hmac: pointer to variables of hmac_handle_t . The variable contains specified register base addresses. |
| Return Value | HAL status |
| Remarks | |

2.14.4.3 hal_hmac_msp_init

Table 2-136 hal_hmac_msp_init API

| | |
|-----------------------------|---|
| Function Prototype | void hal_hmac_msp_init(hmac_handle_t *p_hmac) |
| Function Description | Initialize NVIC interrupts used by the HMAC peripheral. |
| Parameter | p_hmac: pointer to variables of hmac_handle_t |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite the API to initialize NVIC interrupts and DMA channels. |

2.14.4.4 hal_hmac_msp_deinit

Table 2-137 hal_hmac_msp_deinit API

| | |
|-----------------------------|---|
| Function Prototype | void hal_hmac_msp_deinit(hmac_handle_t *p_hmac) |
| Function Description | Deinitialize NVIC interrupts used by the HMAC peripheral. |
| Parameter | p_hmac: pointer to variables of hmac_handle_t |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite the API to deinitialize NVIC interrupts and DMA channels. |

2.14.4.5 hal_hmac_sha256_digest

Table 2-138 hal_hmac_sha256_digest API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_hmac_sha256_digest(hmac_handle_t *p_hmac, uint32_t *p_message, uint32_t number, uint32_t *p_digest, uint32_t timeout) |
| Function Description | Calculate message digests/signatures by using SHA-256 in polling/interrupt/DMA mode. |
| Parameter | p_hmac: pointer to variables of hmac_handle_t |

| | |
|---------------------|---|
| | <p>p_message: pointer to the message to be calculated</p> <p>number: the length of the message to be calculated (in byte). The maximum number is 32768, and the number shall be an integral multiple of 64.</p> <p>p_digest: pointer to calculation results</p> <p>timeout: calculation timeout period (ms)</p> |
| Return Value | HAL status |
| Remarks | If messages to be calculated exceed the maximum number, data segmentation is required for calculation. |

2.14.4.6 hal_hmac_irq_handler

Table 2-139 hal_hmac_irq_handler API

| | |
|-----------------------------|---|
| Function Prototype | void hal_hmac_irq_handler(hmac_handle_t *p_hmac) |
| Function Description | Handle HMAC interrupt requests. |
| Parameter | p_hmac: pointer to variables of hmac_handle_t |
| Return Value | None |
| Remarks | |

2.14.4.7 hal_hmac_done_callback

Table 2-140 hal_hmac_done_callback API

| | |
|-----------------------------|---|
| Function Prototype | void hal_hmac_done_callback(hmac_handle_t *p_hmac) |
| Function Description | HMAC calculation complete callback function |
| Parameter | p_hmac: pointer to variables of hmac_handle_t |
| Return Value | None |
| Remarks | This function is called when the calculation is completed in interrupt/DMA mode. The function is declared empty as weak function. Developers are required to overwrite this function before using it. |

2.14.4.8 hal_hmac_error_callback

Table 2-141 hal_hmac_error_callback API

| | |
|-----------------------------|---|
| Function Prototype | void hal_hmac_error_callback(hmac_handle_t *p_hmac) |
| Function Description | HMAC calculation error callback function |
| Parameter | p_hmac: pointer to variables of hmac_handle_t |
| Return Value | None |

| | |
|----------------|---|
| Remarks | This function is called when an error occurs during calculation in interrupt/DMA mode. The function is declared empty as weak function. Developers are required to overwrite this function before using it. |
|----------------|---|

2.14.4.9 hal_hmac_get_state

Table 2-142 hal_hmac_get_state API

| | |
|-----------------------------|---|
| Function Prototype | hal_hmac_state_t hal_hmac_get_state(hmac_handle_t *p_hmac) |
| Function Description | Get the HMAC operating state. |
| Parameter | p_hmac: pointer to variables of hmac_handle_t |
| Return Value | The HMAC operating state. This parameter can be one of the following values: <ul style="list-style-type: none"> • HAL_HMAC_STATE_RESET (not initialized) • HAL_HMAC_STATE_READY (initialized and ready for use) • HAL_HMAC_STATE_BUSY (busy) • HAL_HMAC_STATE_ERROR (error) • HAL_HMAC_STATE_TIMEOUT (timeout) • HAL_HMAC_STATE_SUSPENDED (suspended) |
| Remarks | |

2.14.4.10 hal_hmac_get_error

Table 2-143 hal_hmac_get_error API

| | |
|-----------------------------|--|
| Function Prototype | unit32_t hal_hmac_get_error(hmac_handle_t *p_hmac) |
| Function Description | Return the HMAC error code. |
| Parameter | p_hmac: pointer to variables of hmac_handle_t |
| Return Value | HMAC error code. This parameter can be one of the following values: <ul style="list-style-type: none"> • HAL_HMAC_ERROR_NONE (no error) • HAL_HMAC_ERROR_TIMEOUT (timeout) • HAL_HMAC_ERROR_TRANSFER (transfer error) • HAL_HMAC_ERROR_INVALID_PARAM (invalid parameter) |
| Remarks | |

2.14.4.11 hal_hmac_set_timeout

Table 2-144 hal_hmac_set_timeout API

| | |
|---------------------------|--|
| Function Prototype | void hal_hmac_set_timeout(hmac_handle_t *p_hmac) |
|---------------------------|--|

| | |
|-----------------------------|---|
| Function Description | Set a timeout period for HMAC operations. |
| Parameter | p_hmac: pointer to variables of hmac_handle_t |
| Return Value | None |
| Remarks | |

2.14.4.12 hal_hmac_suspend_reg

Table 2-145 hal_hmac_suspend_reg API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_hmac_suspend_reg(hmac_handle_t *p_hmac) |
| Function Description | Suspend registers related to HMAC configuration in sleep mode. |
| Parameter | p_hmac: pointer to variables of hmac_handle_t |
| Return Value | HAL status |
| Remarks | |

2.14.4.13 hal_hmac_resume_reg

Table 2-146 hal_hmac_resume_reg API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_hmac_resume_reg(hmac_handle_t *p_hmac) |
| Function Description | Resume registers related to HMAC configuration during wakeup. |
| Parameter | p_hmac: pointer to variables of hmac_handle_t |
| Return Value | HAL status |
| Remarks | |

2.15 HAL PKC Generic Driver

2.15.1 PKC Driver Functionalities

The HAL Public Key Cipher (PKC) driver features the following functionalities:

- Complying with FIPS-180-3 standards; supporting scalar multiplication of P-256 Elliptic Curve algorithm
- Montgomery modular multiplication with configurable data size from 256 bits to 2048 bits
- Partial Montgomery inversion with configurable data size from 256 bits to 2048 bits
- Modular addition operation with configurable data size from 256 bits to 2048 bits
- Modular subtraction operation with configurable data size from 256 bits to 2048 bits
- Modular comparison operation with configurable data size from 256 bits to 2048 bits
- Modular left shift operation with configurable data size from 256 bits to 2048 bits
- Big data multiplication with configurable data size from 256 bits to 1024 bits

- Big data addition with configurable data size from 256 bits to 2048 bits
- Dummy multiplication available for hardware
- Random clock scrambling
- Adopting the single-port RAM with the size of 1280 bytes and the data width of 32 bits; the RAM can be read by MCU and DMA
- Two operation approaches: polling and interrupt
- Aborting operations in interrupt mode
- Operation complete, error, overflow, and abort complete interrupt callback functions
- Getting operating state and error code of PKC driver
- Timeout settings

2.15.2 How to Use PKC Driver

Developers can use the PKC driver in the following scenarios:

1. Declare a `pkc_handle_t` handle structure variable, for example: `pkc_handle_t pkc_handle`.
2. Initialize the PKC low-level resources by overwriting `hal_pkc_msp_init()`:
 - (1). Call `__HAL_PKC_RESET()` to reset the PKC module.
 - (2). If interrupt APIs are required, developers need to call related NVIC APIs:
 - Configure the PKC interrupt priority by calling `hal_nvic_set_priority()`.
 - Enable NVIC interrupts for the PKC driver by calling `hal_nvic_enable_irq()`.
3. Configure parameters for the init structure of `pkc_handle`, including data bit width, security mode, generation function of registered random number, and Elliptic Curve Cryptography (ECC).
4. Configure PKC registers by calling `hal_pkc_init()`. During configuration, `hal_pkc_init()` automatically calls the overwritten `hal_pkc_msp_init()`, to initialize NVIC interrupts and other low-level resources for PKC.
5. Developers can call corresponding APIs for mathematics in actual use. The PKC driver supports two operation approaches: polling and interrupt.

2.15.3 PKC Driver Structures and Defines

2.15.3.1 ecc_point_t

The ECC point description structure `ecc_point_t` of the PKC driver is defined below:

Table 2-147 `ecc_point_t` structure

| Data Field | Field Description | Value |
|---|--------------------------|--------------------|
| <code>uint32_t X[ECC_U32_LENGTH]</code> | X position of ECC point. | 0 to $2^{256} - 1$ |

| Data Field | Field Description | Value |
|----------------------------|--------------------------|--------------------|
| uint32_t Y[ECC_U32_LENGTH] | Y position of ECC point. | 0 to $2^{256} - 1$ |

2.15.3.2 ecc_curve_init_t

The ECC description structure `ecc_curve_init_t` of the PKC driver is defined below:

Table 2-148 `ecc_curve_init_t` structure

| Data Field | Field Description | Value |
|-----------------------------------|----------------------|--|
| uint32_t A[ECC_U32_LENGTH] | Operand A array | 0 to $2^{256} - 1$ |
| uint32_t B[ECC_U32_LENGTH] | Operand B array | 0 to $2^{256} - 1$ |
| uint32_t P[ECC_U32_LENGTH] | Prime number P array | 0 to $2^{256} - 1$ |
| uint32_t PRSquare[ECC_U32_LENGTH] | PRSquare array | $R^2 \bmod P$, where $R = 2^{256}$ |
| uint32_t ConstP | ConstP array | Montgomery constant of prime number P |
| uint32_t N[ECC_U32_LENGTH] | Prime number N array | 0 to $2^{256} - 1$ |
| uint32_t NRSquare[ECC_U32_LENGTH] | NRSquare array | $R^2 \bmod N$, where $R = 2^{256}$ |
| uint32_t ConstN | ConstN array | Montgomery constant of prime number N |
| uint32_t H | Parameter H | 0 to $2^{32} - 1$ |
| ll_ecc_point_t G | ECC point | See " Section 3.10.1.1 ll_ecc_point_t ". |

2.15.3.3 pkc_init_t

The initialization structure `pkc_init_t` of the PKC driver is defined below:

Table 2-149 `pkc_init_t` structure

| Data Field | Field Description | Value |
|--|--|---|
| <code>ecc_curve_init_t *p_ecc_curve</code> | Pointer to the elliptic curve description type | See " Section 2.15.3.2 ecc_curve_init_t ". |
| uint32_t data_bits | Calculation data bit width | 256 to 2048 |
| uint32_t secure_mode | To enable security mode or not | This parameter can be one of the following values: <ul style="list-style-type: none"> PKC_SECURE_MODE_DISABLE (disable) PKC_SECURE_MODE_ENABLE (enable) |
| uint32_t (*random_func)(void) | Specified random number generation function | Function pointer |

2.15.3.4 pkc_handle_t

The handle structure pkc_handle_t of the PKC driver is defined below:

Table 2-150 pkc_handle_t structure

| Data Field | Field Description | Value |
|-----------------------------|---|--|
| pkc_regs_t *p_instance | PKC peripheral instance | PKC |
| pkc_init_t init | Initialization structure | See " Section 2.15.3.3 pkc_init_t ". |
| void *p_result | Pointer to data operation results | Developers need to specify this parameter each time before calling an operation API. |
| uint32_t *k_kout | Pointer to Montgomery inverse operation results | Developers need to specify this parameter each time before calling an operation API. |
| uint32_t shift_count | Shift count (managed by PKC driver and initialization by developers not required) | N/A |
| __IO hal_lock_t lock | PKC lock (managed by PKC driver and initialization by developers not required) | N/A |
| __IO hal_qspi_state_t state | PKC operating state (initialization by developers not required) | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • HAL_PKC_STATE_RESET (not initialized) • HAL_PKC_STATE_READY (initialized and ready for use) • HAL_PKC_STATE_BUSY (busy) • HAL_PKC_STATE_ERROR (error) • HAL_PKC_STATE_TIMEOUT (timeout) |
| __IO uint32_t error_code | PKC error code (initialization by developers not required) | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • HAL_PKC_ERROR_NONE (no error) • HAL_PKC_ERROR_TIMEOUT (timeout) • HAL_PKC_ERROR_TRANSFER (transfer error) • HAL_PKC_ERROR_OVERFLOW (overflow error) • HAL_PKC_ERROR_INVALID_PARAM (invalid parameter) • HAL_PKC_ERROR_INVERSE_K (coefficient K error in output parameters for inverse operation) • HAL_PKC_ERROR_IRREVERSIBLE (irreversible input parameter for inverse operation) |
| uint32_t timeout | PKC timeout period (initialization by developers not required) | N/A |

| Data Field | Field Description | Value |
|-----------------------|--|-------|
| uint32_t retention[1] | PKC register information (managed by PKC driver and initialization by developers not required) | N/A |

2.15.3.5 pkc_ecc_point_multi_t

The ECC point multiplication structure `pkc_ecc_point_multi_t` of the PKC driver is defined below:

Table 2-151 `pkc_ecc_point_multi_t` structure

| Data Field | Field Description | Value |
|--------------------------|--------------------------|---|
| uint32_t *p_K | Input parameter K | 0 to $2^{256} - 1$ |
| ecc_point_t *p_ecc_point | Input parameter ECCPoint | See " Section 2.15.3.1 ecc_point_t ". |

2.15.3.6 pkc_rsa_modular_exponent_t

The Rivest–Shamir–Adleman (RSA) modular exponentiation structure `pkc_rsa_modular_exponent_t` of the PKC driver is defined below:

Table 2-152 `pkc_rsa_modular_exponent_t` structure

| Data Field | Field Description | Value |
|-----------------|----------------------------------|--|
| uint32_t *p_A | Operand A (input parameter) | Point to <code>data_bits</code> data |
| uint32_t *p_B | Operand B (input parameter) | Point to <code>data_bits</code> data |
| uint32_t *p_P | Prime number P (input parameter) | Point to <code>data_bits</code> data |
| uint32_t *p_R2 | Input parameter R2 | $R2 = R^2 \bmod P$, $R = 2^{\text{data_bits}}$ |
| uint32_t ConstP | Input parameter ConstP | Montgomery constant of prime number P |

2.15.3.7 pkc_modular_add_t

The modular addition structure `pkc_modular_add_t` of the PKC driver is defined below:

Table 2-153 `pkc_modular_add_t` structure

| Data Field | Field Description | Value |
|---------------|----------------------------------|--------------------------------------|
| uint32_t *p_A | Operand A (input parameter) | Point to <code>data_bits</code> data |
| uint32_t *p_B | Operand B (input parameter) | Point to <code>data_bits</code> data |
| uint32_t *p_P | Prime number P (input parameter) | Point to <code>data_bits</code> data |

2.15.3.8 pkc_modular_sub_t

The modular subtraction structure `pkc_modular_sub_t` of the PKC driver is defined below:

Table 2-154 `pkc_modular_sub_t` structure

| Data Field | Field Description | Value |
|----------------------------|----------------------------------|--------------------------------------|
| <code>uint32_t *p_A</code> | Operand A (input parameter) | Point to <code>data_bits</code> data |
| <code>uint32_t *p_B</code> | Operand B (input parameter) | Point to <code>data_bits</code> data |
| <code>uint32_t *p_P</code> | Prime number P (input parameter) | Point to <code>data_bits</code> data |

2.15.3.9 `pkc_modular_shift_t`

The modular left shift operation structure `pkc_modular_shift_t` of the PKC driver is defined below:

Table 2-155 `pkc_modular_shift_t` structure

| Data Field | Field Description | Value |
|----------------------------------|-----------------------------------|--------------------------------------|
| <code>uint32_t *p_A</code> | Operand A (input parameter) | Point to <code>data_bits</code> data |
| <code>uint32_t shift_bits</code> | Left shift bits (input parameter) | 1 to <code>data_bits</code> |
| <code>uint32_t *p_P</code> | Prime number P (input parameter) | Point to <code>data_bits</code> data |

2.15.3.10 `pkc_modular_compare_t`

The modular comparison operation structure `pkc_modular_compare_t` of the PKC driver is defined below:

Table 2-156 `pkc_modular_compare_t` structure

| Data Field | Field Description | Value |
|----------------------------|----------------------------------|--------------------------------------|
| <code>uint32_t *p_A</code> | Operand A (input parameter) | Point to <code>data_bits</code> data |
| <code>uint32_t *p_P</code> | Prime number P (input parameter) | Point to <code>data_bits</code> data |

2.15.3.11 `pkc_montgomery_multi_t`

The Montgomery multiplication structure `pkc_montgomery_multi_t` of the PKC driver is defined below:

Table 2-157 `pkc_montgomery_multi_t` structure

| Data Field | Field Description | Value |
|------------------------------|----------------------------------|---------------------------------------|
| <code>uint32_t *p_A</code> | Operand A (input parameter) | Point to <code>data_bits</code> data |
| <code>uint32_t *p_B</code> | Operand B (input parameter) | Point to <code>data_bits</code> data |
| <code>uint32_t *p_P</code> | Prime number P (input parameter) | Point to <code>data_bits</code> data |
| <code>uint32_t ConstP</code> | Input parameter ConstP | Montgomery constant of prime number P |

2.15.3.12 `pkc_montgomery_inversion_t`

The Montgomery inversion structure `pkc_montgomery_inversion_t` of the PKC driver is defined below:

Table 2-158 pkc_montgomery_inversion_t structure

| Data Field | Field Description | Value |
|---------------|----------------------------------|-------------------------|
| uint32_t *p_A | Operand A (input parameter) | Point to data_bits data |
| uint32_t *p_P | Prime number P (input parameter) | Point to data_bits data |

2.15.3.13 pkc_big_number_multi_t

The big data multiplication structure pkc_big_number_multi_t of the PKC driver is defined below:

Table 2-159 pkc_big_number_multi_t structure

| Data Field | Field Description | Value |
|---------------|-----------------------------|-------------------------|
| uint32_t *p_A | Operand A (input parameter) | Point to data_bits data |
| uint32_t *p_B | Operand B (input parameter) | Point to data_bits data |

2.15.3.14 pkc_big_number_add_t

The big data addition structure pkc_big_number_add_t of the PKC driver is defined below:

Table 2-160 pkc_big_number_add_t structure

| Data Field | Field Description | Value |
|---------------|-----------------------------|-------------------------|
| uint32_t *p_A | Operand A (input parameter) | Point to data_bits data |
| uint32_t *p_B | Operand B (input parameter) | Point to data_bits data |

2.15.4 PKC Driver APIs

The PKC driver APIs are listed in the table below:

Table 2-161 PKC driver APIs

| API Type | API Name | Description |
|----------------|--------------------------------|---|
| Initialization | hal_pkc_init() | Initialize the PKC peripheral, and configure data bit width and other parameters. |
| | hal_pkc_deinit() | Deinitialize the PKC peripheral. |
| | hal_pkc_msp_init() | Initialize NVIC interrupts used by the PKC peripheral. |
| | hal_pkc_msp_deinit() | Deinitialize NVIC interrupts used by the PKC peripheral. |
| I/O operation | hal_pkc_rsa_modular_exponent() | RSA modular exponentiation in polling mode |
| | hal_pkc_ecc_point_multi() | ECC point multiplication in polling mode |
| | hal_pkc_modular_add() | Modular addition in polling mode |

| API Type | API Name | Description |
|---------------------------------|-----------------------------------|---|
| | hal_pkc_modular_sub() | Modular subtraction in polling mode |
| | hal_pkc_modular_left_shift() | Modular left shift operation in polling mode |
| | hal_pkc_modular_compare() | Modular comparison operation in polling mode |
| | hal_pkc_montgomery_multi() | Montgomery multiplication in polling mode |
| | hal_pkc_montgomery_inversion() | Montgomery inversion in polling mode |
| | hal_pkc_big_number_multi() | Big data multiplication in polling mode |
| | hal_pkc_big_number_add() | Big data addition in polling mode |
| | hal_pkc_ecc_point_multi_it() | ECC point multiplication in interrupt mode |
| | hal_pkc_modular_add_it() | Modular addition in interrupt mode |
| | hal_pkc_modular_sub_it() | Modular subtraction in interrupt mode |
| | hal_pkc_modular_left_shift_it() | Modular left shift operation in interrupt mode |
| | hal_pkc_modular_compare_it() | Modular comparison operation in interrupt mode |
| | hal_pkc_montgomery_multi_it() | Montgomery multiplication in interrupt mode |
| | hal_pkc_montgomery_inversion_it() | Montgomery inversion in interrupt mode |
| | hal_pkc_big_number_multi_it() | Big data multiplication in interrupt mode |
| | hal_pkc_big_number_add_it() | Big data addition in interrupt mode |
| Interrupt handling and callback | hal_pkc_irq_handler() | Interrupt handler |
| | hal_pkc_done_callback() | Operation complete interrupt callback function |
| | hal_pkc_error_callback() | Error interrupt callback function |
| | hal_pkc_overflow_callback() | Operation overflow interrupt callback function |
| State and error | hal_pkc_get_state() | Get the driver operating state. |
| | hal_pkc_get_error() | Get error code. |
| Control | hal_pkc_set_timeout() | Set a timeout period. |
| Sleep | hal_pkc_suspend_reg() | Suspend registers related to PKC configuration in sleep mode. |
| | hal_pkc_resume_reg() | Resume registers related to PKC configuration during wakeup. |

The sections below elaborate on these APIs.

2.15.4.1 hal_pkc_init

Table 2-162 hal_pkc_init API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_pkc_init(pkc_handle_t *p_pkc) |
| Function Description | Initialize the PKC peripheral and related handles according to parameters of pkc_init_t . |

| | |
|---------------------|---|
| Parameter | p_pkc: pointer to variables of pkc_handle_t . The variable contains the configuration information of a specified PKC. |
| Return Value | HAL status |
| Remarks | |

2.15.4.2 hal_pkc_deinit

Table 2-163 hal_pkc_deinit API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_pkc_deinit(pkc_handle_t *p_pkc) |
| Function Description | Deinitialize the PKC peripheral. |
| Parameter | p_pkc: pointer to variables of pkc_handle_t . The variable contains the configuration information of a specified PKC. |
| Return Value | HAL status |
| Remarks | |

2.15.4.3 hal_pkc_msp_init

Table 2-164 hal_pkc_msp_init API

| | |
|-----------------------------|--|
| Function Prototype | void hal_pkc_msp_init(pkc_handle_t *p_pkc) |
| Function Description | Initialize NVIC interrupts used by the PKC peripheral. |
| Parameter | p_pkc: pointer to variables of pkc_handle_t |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite the API to initialize NVIC interrupts. |

2.15.4.4 hal_pkc_msp_deinit

Table 2-165 hal_pkc_msp_deinit API

| | |
|-----------------------------|--|
| Function Prototype | void hal_pkc_msp_deinit(pkc_handle_t *p_pkc) |
| Function Description | Deinitialize NVIC interrupts used by the PKC peripheral. |
| Parameter | p_pkc: pointer to variables of pkc_handle_t |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite the API to deinitialize NVIC interrupts. |

2.15.4.5 hal_pkc_rsa_modular_exponent

Table 2-166 hal_pkc_rsa_modular_exponent API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_pkc_rsa_modular_exponent(pkc_handle_t *p_pkc, pkc_rsa_modular_exponent_t *p_input, uint32_t timeout) |
| Function Description | Perform RSA modular exponentiation: Result = A^B mod P; maximum bit for the operand and the result: 2048 bits; in polling mode |
| Parameter | p_pkc: pointer to variables of pkc_handle_t p_input: pointer to variables of pkc_rsa_modular_exponent_t timeout: timeout period |
| Return Value | HAL status |
| Remarks | |

2.15.4.6 hal_pkc_ecc_point_multi

Table 2-167 hal_pkc_ecc_point_multi API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_pkc_ecc_point_multi(pkc_handle_t *p_pkc, pkc_ecc_point_multi_t *p_input, uint32_t timeout) |
| Function Description | Perform ECC point multiplication: Result = K x Point; maximum bit for the operand and the result: 2048 bits; in polling mode |
| Parameter | p_pkc: pointer to variables of pkc_handle_t p_input: pointer to variables of pkc_ecc_point_multi_t timeout: timeout period |
| Return Value | HAL status |
| Remarks | |

2.15.4.7 hal_pkc_ecc_point_multi_it

Table 2-168 hal_pkc_ecc_point_multi_it API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_pkc_ecc_point_multi_it(pkc_handle_t *p_pkc, pkc_ecc_point_multi_t *p_input) |
| Function Description | Perform ECC point multiplication: Result = K x Point; maximum bit for the operand and the result: 2048 bits; in interrupt mode |
| Parameter | p_pkc: pointer to variables of pkc_handle_t p_input: pointer to variables of pkc_ecc_point_multi_t |
| Return Value | HAL status |
| Remarks | |

2.15.4.8 hal_pkc_modular_add

Table 2-169 hal_pkc_modular_add API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_pkc_modular_add(pkc_handle_t *p_pkc, pkc_modular_add_t *p_input, uint32_t timeout) |
| Function Description | Perform modular addition: Result = (A + B) mod P; maximum bit for the operand and the result: 2048 bits; in polling mode |
| Parameter | p_pkc: pointer to variables of pkc_handle_t p_input: pointer to variables of pkc_modular_add_t timeout: timeout period |
| Return Value | HAL status |
| Remarks | |

2.15.4.9 hal_pkc_modular_add_it

Table 2-170 hal_pkc_modular_add_it API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_pkc_modular_add_it(pkc_handle_t *p_pkc, pkc_modular_add_t *p_input) |
| Function Description | Perform modular addition: Result = (A + B) mod P; maximum bit for the operand and the result: 2048 bits; in interrupt mode |
| Parameter | p_pkc: pointer to variables of pkc_handle_t p_input: pointer to variables of pkc_modular_add_t |
| Return Value | HAL status |
| Remarks | |

2.15.4.10 hal_pkc_modular_sub

Table 2-171 hal_pkc_modular_sub API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_pkc_modular_sub(pkc_handle_t *p_pkc, pkc_modular_sub_t *p_input, uint32_t timeout) |
| Function Description | Perform modular subtraction: Result = (A - B) mod P; maximum bit for the operand and the result: 2048 bits; in polling mode |
| Parameter | p_pkc: pointer to variables of pkc_handle_t p_input: pointer to variables of pkc_modular_sub_t timeout: timeout period |
| Return Value | HAL status |
| Remarks | |

2.15.4.11 hal_pkc_modular_sub_it

Table 2-172 hal_pkc_modular_sub_it API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_pkc_modular_sub_it(pkc_handle_t *p_pkc, pkc_modular_sub_t *p_input) |
| Function Description | Perform modular subtraction: Result = (A - B) mod P; maximum bit for the operand and the result: 2048 bits; in interrupt mode |
| Parameter | p_pkc: pointer to variables of pkc_handle_t p_input: pointer to variables of pkc_modular_sub_t |
| Return Value | HAL status |
| Remarks | |

2.15.4.12 hal_pkc_modular_left_shift

Table 2-173 hal_pkc_modular_left_shift API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_pkc_modular_left_shift(pkc_handle_t *p_pkc, pkc_modular_shift_t *p_input, uint32_t timeout) |
| Function Description | Perform modular left shift operation: Result = (A << ShiftBits) mod P; maximum bit for the operand and the result: 2048 bits; in polling mode |
| Parameter | p_pkc: pointer to variables of pkc_handle_t p_input: pointer to variables of pkc_modular_shift_t timeout: timeout period |
| Return Value | HAL status |
| Remarks | |

2.15.4.13 hal_pkc_modular_left_shift_it

Table 2-174 hal_pkc_modular_left_shift_it API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_pkc_modular_left_shift_it(pkc_handle_t *p_pkc, pkc_modular_shift_t *p_input) |
| Function Description | Perform modular left shift operation: Result = (A << ShiftBits) mod P; maximum bit for the operand and the result: 2048 bits; in interrupt mode |
| Parameter | p_pkc: pointer to variables of pkc_handle_t p_input: pointer to variables of pkc_modular_shift_t |
| Return Value | HAL status |
| Remarks | |

2.15.4.14 hal_pkc_modular_compare

Table 2-175 hal_pkc_modular_compare API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_pkc_modular_compare(pkc_handle_t *p_pkc, pkc_modular_compare_t *p_input, uint32_t timeout) |
| Function Description | Perform modular comparison operation: Result = A mod P; maximum bit for the operand and the result: 2048 bits; in polling mode |
| Parameter | p_pkc: pointer to variables of pkc_handle_t p_input: pointer to variables of pkc_modular_compare_t timeout: timeout period |
| Return Value | HAL status |
| Remarks | |

2.15.4.15 hal_pkc_modular_compare_it

Table 2-176 hal_pkc_modular_compare_it API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_pkc_modular_compare_it(pkc_handle_t *p_pkc, pkc_modular_compare_t *p_input) |
| Function Description | Perform modular comparison operation: Result = A mod P; maximum bit for the operand and the result: 2048 bits; in interrupt mode |
| Parameter | p_pkc: pointer to variables of pkc_handle_t p_input: pointer to variables of pkc_modular_compare_t |
| Return Value | HAL status |
| Remarks | |

2.15.4.16 hal_pkc_montgomery_multi

Table 2-177 hal_pkc_montgomery_multi API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_pkc_montgomery_multi(pkc_handle_t *p_pkc, pkc_montgomery_multi_t *p_input, uint32_t timeout) |
| Function Description | Perform modular multiplication: Result = A x B mod P; maximum bit for the operand and the result: 2048 bits; in polling mode |
| Parameter | p_pkc: pointer to variables of pkc_handle_t p_input: pointer to variables of pkc_montgomery_multi_t timeout: timeout period |
| Return Value | HAL status |
| Remarks | |

2.15.4.17 hal_pkc_montgomery_multi_it

Table 2-178 hal_pkc_montgomery_multi_it API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_pkc_montgomery_multi_it(pkc_handle_t *p_pkc, pkc_montgomery_multi_t *p_input) |
| Function Description | Perform modular multiplication: Result = $A \times B \pmod{P}$; maximum bit for the operand and the result: 2048 bits; in interrupt mode |
| Parameter | p_pkc: pointer to variables of pkc_handle_t p_input: pointer to variables of pkc_montgomery_multi_t |
| Return Value | HAL status |
| Remarks | |

2.15.4.18 hal_pkc_montgomery_inversion

Table 2-179 hal_pkc_montgomery_inversion API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_pkc_montgomery_inversion(pkc_handle_t *p_pkc, pkc_montgomery_inversion_t *p_input, uint32_t *p_K, uint32_t timeout) |
| Function Description | Perform modular inverse operation: Result = $A^{(-1)} \pmod{P}$; maximum bit for the operand and the result: 2048 bits; in polling mode |
| Parameter | p_pkc: pointer to variables of pkc_handle_t p_input: pointer to variables of pkc_montgomery_inversion_t p_K: pointer to the output parameter K timeout: timeout period |
| Return Value | HAL status |
| Remarks | |

2.15.4.19 hal_pkc_montgomery_inversion_it

Table 2-180 hal_pkc_montgomery_inversion_it API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_pkc_montgomery_inversion_it(pkc_handle_t *p_pkc, pkc_montgomery_inversion_t *p_input, uint32_t *p_K) |
| Function Description | Perform modular inverse operation: Result = $A^{(-1)} \pmod{P}$; maximum bit for the operand and the result: 2048 bits; in interrupt mode |
| Parameter | p_pkc: pointer to variables of pkc_handle_t p_input: pointer to variables of pkc_montgomery_inversion_t p_K: pointer to the output parameter K |
| Return Value | HAL status |

| | |
|---------|--|
| Remarks | |
|---------|--|

2.15.4.20 hal_pkc_big_number_multi

Table 2-181 hal_pkc_big_number_multi API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_pkc_big_number_multi(pkc_handle_t *p_pkc, pkc_big_number_multi_t *p_input, uint32_t timeout) |
| Function Description | Perform big data multiplication: Result = A + B; maximum bit for the operand and the result: 1024 bits; in polling mode |
| Parameter | p_pkc: pointer to variables of pkc_handle_t p_input: pointer to variables of pkc_big_number_multi_t timeout: timeout period |
| Return Value | HAL status |
| Remarks | |

2.15.4.21 hal_pkc_big_number_multi_it

Table 2-182 hal_pkc_big_number_multi_it API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_pkc_big_number_multi_it(pkc_handle_t *p_pkc, pkc_big_number_multi_t *p_input) |
| Function Description | Perform big data multiplication: Result = A x B; maximum bit for the operand: 1024 bits; for the result: 2048 bits; in interrupt mode |
| Parameter | p_pkc: pointer to variables of pkc_handle_t p_input: pointer to variables of pkc_big_number_multi_t |
| Return Value | HAL status |
| Remarks | |

2.15.4.22 hal_pkc_big_number_add

Table 2-183 hal_pkc_big_number_add API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_pkc_big_number_add(pkc_handle_t *p_pkc, pkc_big_number_add_t *p_input, uint32_t timeout) |
| Function Description | Perform big data addition: Result = A + B; maximum bit for the operand and the result: 2048 bits; in polling mode |
| Parameter | p_pkc: pointer to variables of pkc_handle_t p_input: pointer to variables of pkc_big_number_add_t timeout: timeout period |

| | |
|--------------|------------|
| Return Value | HAL status |
| Remarks | |

2.15.4.23 hal_pkc_big_number_add_it

Table 2-184 hal_pkc_big_number_add_it API

| | |
|----------------------|--|
| Function Prototype | hal_status_t hal_pkc_big_number_add_it(pkc_handle_t *p_pkc, pkc_big_number_add_t *p_input) |
| Function Description | Perform big data addition: Result = A + B; maximum bit for the operand and the result: 2048 bits; in interrupt mode |
| Parameter | p_pkc: pointer to variables of pkc_handle_t p_input: pointer to variables of pkc_big_number_add_t |
| Return Value | HAL status |
| Remarks | |

2.15.4.24 hal_pkc_irq_handler

Table 2-185 hal_pkc_irq_handler API

| | |
|----------------------|---|
| Function Prototype | void hal_pkc_irq_handler(pkc_handle_t *p_pkc) |
| Function Description | Handle PKC interrupt requests. |
| Parameter | p_pkc: pointer to variables of pkc_handle_t |
| Return Value | None |
| Remarks | |

2.15.4.25 hal_pkc_done_callback

Table 2-186 hal_pkc_done_callback API

| | |
|----------------------|--|
| Function Prototype | void hal_pkc_done_callback(pkc_handle_t *p_pkc) |
| Function Description | PKC operation complete callback function |
| Parameter | p_pkc: pointer to variables of pkc_handle_t |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite this function before using it. |

2.15.4.26 hal_pkc_error_callback

Table 2-187 hal_pkc_error_callback API

| | |
|-----------------------------|--|
| Function Prototype | void hal_pkc_error_callback(pkc_handle_t *p_pkc) |
| Function Description | PKC operation error callback function |
| Parameter | p_pkc: pointer to variables of pkc_handle_t |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite this function before using it. |

2.15.4.27 hal_pkc_overflow_callback

Table 2-188 hal_pkc_overflow_callback API

| | |
|-----------------------------|--|
| Function Prototype | void hal_pkc_overflow_callback(pkc_handle_t *p_pkc) |
| Function Description | PKC big data multiplication/addition overflow callback function |
| Parameter | p_pkc: pointer to variables of pkc_handle_t |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite this function before using it. |

2.15.4.28 hal_pkc_get_state

Table 2-189 hal_pkc_get_state API

| | |
|-----------------------------|---|
| Function Prototype | hal_pkc_state_t hal_pkc_get_state(pkc_handle_t *p_pkc) |
| Function Description | Get the PKC operating state. |
| Parameter | p_pkc: pointer to variables of pkc_handle_t |
| Return Value | PKC operating state. This parameter can be one of the following values: <ul style="list-style-type: none"> • HAL_PKC_STATE_RESET (not initialized) • HAL_PKC_STATE_READY (initialized and ready for use) • HAL_PKC_STATE_BUSY (busy) • HAL_PKC_STATE_ERROR (error) • HAL_PKC_STATE_TIMEOUT (timeout) |
| Remarks | |

2.15.4.29 hal_pkc_get_error

Table 2-190 hal_pkc_get_error API

| | |
|---------------------------|---|
| Function Prototype | uint32_t hal_pkc_get_error(pkc_handle_t *p_pkc) |
|---------------------------|---|

| | |
|-----------------------------|--|
| Function Description | Return the PKC error code. |
| Parameter | p_pkc: pointer to variables of pkc_handle_t |
| Return Value | <p>PKC error code. This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • HAL_PKC_ERROR_NONE (no error) • HAL_PKC_ERROR_TIMEOUT (timeout) • HAL_PKC_ERROR_TRANSFER (transfer error) • HAL_PKC_ERROR_OVERFLOW (overflow error) • HAL_PKC_ERROR_INVALID_PARAM (invalid parameter) • HAL_PKC_ERROR_INVERSE_K (coefficient K error in output parameters for inverse operation) • HAL_PKC_ERROR_IRREVERSIBLE (irreversible input parameter for inverse operation) |
| Remarks | |

2.15.4.30 hal_pkc_set_timeout

Table 2-191 hal_pkc_set_timeout API

| | |
|-----------------------------|---|
| Function Prototype | void hal_pkc_set_timeout(pkc_handle_t *p_pkc , uint32_t timeout) |
| Function Description | Set a timeout period for PKC operations. |
| Parameter | <p>p_pkc: pointer to variables of pkc_handle_t</p> <p>timeout: operation timeout period</p> |
| Return Value | None |
| Remarks | |

2.15.4.31 hal_pkc_suspend_reg

Table 2-192 hal_pkc_suspend_reg API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_pkc_suspend_reg(pkc_handle_t *p_pkc) |
| Function Description | Suspend registers related to PKC configuration in sleep mode. |
| Parameter | p_pkc: pointer to variables of pkc_handle_t |
| Return Value | <p>PKC operating state. This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • HAL_PKC_STATE_RESET (not initialized) • HAL_PKC_STATE_READY (initialized and ready for use) • HAL_PKC_STATE_BUSY (busy) • HAL_PKC_STATE_ERROR (error) • HAL_PKC_STATE_TIMEOUT (timeout) |
| Remarks | |

2.15.4.32 hal_pkc_resume_reg

Table 2-193 hal_pkc_resume_reg API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_pkc_resume_reg(pkc_handle_t *p_pkc) |
| Function Description | Resume registers related to PKC configuration during wakeup. |
| Parameter | p_pkc: pointer to variables of pkc_handle_t |
| Return Value | PKC operating state. This parameter can be one of the following values: <ul style="list-style-type: none"> • HAL_PKC_STATE_RESET (not initialized) • HAL_PKC_STATE_READY (initialized and ready for use) • HAL_PKC_STATE_BUSY (busy) • HAL_PKC_STATE_ERROR (error) • HAL_PKC_STATE_TIMEOUT (timeout) |
| Remarks | |

2.16 HAL I2C Generic Driver

2.16.1 I2C Driver Functionalities

The HAL Inter-integrated Circuit (I2C) driver features the following functionalities:

- Data read and write in standard mode (0 to 100 Kb/s), fast mode (≤ 400 Kb/s), fast plus mode (≤ 1000 Kb/s), and high-speed mode (≤ 2.8 Mb/s)
- Automatic switching between the master/slave mode
- 7-bit or 10-bit addressing mode
- 7-bit or 10-bit hybrid addressing mode
- Read from and write to external storage devices
- Three operating modes: polling, interrupt, and DMA
- Aborting data TX and RX/read and write in interrupt/DMA mode
- TX and RX complete interrupt callback function in master/slave mode
- Write complete and read complete interrupt callback functions in memory mode
- Abort complete and I/O error interrupt callback functions
- Getting I2C mode, operating state, and error code of I2C driver

2.16.2 How to Use I2C Driver

Developers can use the I2C driver in the following scenarios:

1. Define a structure variable of `i2c_handle_t`, such as `i2c_handle_t i2c_handle` (`i2c_handle_t` structure is defined in the I2C driver. Developers shall define a variable for this type of handle structure before running it.)
2. Initialize the I2C low-level resources by overwriting `hal_i2c_msp_init()`:
 - (1). Configure corresponding I2C GPIOs for multiplexing functionalities and enable pull-up resistors.
 - (2). If I/O operation APIs in interrupt mode or DMA mode are required, developers need to call related NVIC APIs:
 - Configure the I2C interrupt priority by calling `hal_nvic_set_priority()`.
 - Enable NVIC interrupts for the I2C driver by calling `hal_nvic_enable_irq()`.
 - (3). Configure the DMA channels before using I/O APIs in DMA mode.
 - Define `dma_handle_t` handle structure variables for transmission/reception, such as `dma_handle_t dma_tx` and `dma_handle_t dma_rx`.
 - Configure parameters of DMA handle (`dma_tx` and `dma_rx`), for example, specifying TX or RX channels.
 - Point `p_dmatx` and `p_dmarx` (in I2C handler structure variables) to `dma_tx` and `dma_rx`, the initialized variables in DMA handle.
 - Configure the DMA interrupt priority, and enable NVIC interrupts for DMA.
3. Configure data transfer rate, local device address, addressing mode, and advertising address monitoring mode in the I2C initialization structure.
4. Configure I2C registers by calling `hal_i2c_init()`. During configuration, `hal_i2c_init()` automatically calls the overwritten `hal_i2c_msp_init()`, to initialize GPIOs and other low-level resources for I2C.
5. The I2C driver provides three modes for I2C I/O operations (data read/write or memory read/write): polling, interrupt, and DMA.

2.16.2.1 I/O Read and Write in Polling Mode

1. Transmit a large volume of data as a master in polling mode by running `hal_i2c_master_transmit()`.
2. Receive a large volume of data as a master in polling mode by running `hal_i2c_master_receive()`.
3. Transmit a large volume of data as a slave in polling mode by running `hal_i2c_slave_transmit()`.
4. Receive a large volume of data as a slave in polling mode by running `hal_i2c_slave_receive()`.

2.16.2.2 I/O Memory Read and Write in Polling Mode

1. Write a large volume of data to a specified address in polling mode by running `hal_i2c_mem_write()`.
2. Read a large volume of data from a specified address in polling mode by running `hal_i2c_mem_read()`.

2.16.2.3 I/O Read and Write in Interrupt Mode

1. Transmit a large volume of data as a master in interrupt mode by running `hal_i2c_master_transmit_it()`. When a transmission completes, `hal_i2c_master_tx_cplt_callback()` will be called. You can overwrite the callback for certain operations.
2. Receive a large volume of data as a master in interrupt mode by running `hal_i2c_master_receive_it()`. When a reception completes, `hal_i2c_master_rx_cplt_callback()` will be called. You can overwrite the callback for certain operations.
3. Transmit a large volume of data as a slave in interrupt mode by running `hal_i2c_slave_transmit_it()`. When a transmission completes, `hal_i2c_slave_tx_cplt_callback()` will be called. You can overwrite the callback for certain operations.
4. Receive a large volume of data as a slave in interrupt mode by running `hal_i2c_slave_receive_it()`. When a reception completes, `hal_i2c_slave_rx_cplt_callback()` will be called. You can overwrite the callback for certain operations.
5. If errors occur during data transmission/reception, `hal_i2c_error_callback()` will be called. You can overwrite the callback for certain operations.
6. Run `hal_i2c_master_abort_it()` to abort data transmission/reception as a master. If the abort completes, `hal_i2c_abort_cplt_callback()` will be called. You can overwrite the callback for certain operations.

2.16.2.4 I/O Memory Read and Write in Interrupt Mode

1. Write a large volume of data to a specified address in interrupt mode by running `hal_i2c_mem_write_it()`. When a write completes, `hal_i2c_mem_tx_cplt_callback()` will be called. You can overwrite the callback for certain operations.
2. Read a large volume of data from a specified address in interrupt mode by running `hal_i2c_mem_read_it()`. When a read completes, `hal_i2c_mem_rx_cplt_callback()` will be called. You can overwrite the callback for certain operations.
3. If errors occur during data transmission/reception, `hal_i2c_error_callback()` will be called. You can overwrite the callback for certain operations.

2.16.2.5 I/O Read and Write in DMA Mode

1. Transmit a large volume of data as a master in DMA mode by running `hal_i2c_master_transmit_dma()`. When a transmission completes, `hal_i2c_master_tx_cplt_callback()` will be called. You can overwrite the callback for certain operations.
2. Receive a large volume of data as a master in DMA mode by running `hal_i2c_master_receive_dma()`. When a reception completes, `hal_i2c_master_rx_cplt_callback()` will be called. You can overwrite the callback for certain operations.
3. Transmit a large volume of data as a slave in DMA mode by running `hal_i2c_slave_transmit_dma()`. When a transmission completes, `hal_i2c_slave_tx_cplt_callback()` will be called. You can overwrite the callback for certain operations.

4. Receive a large volume of data as a slave in DMA mode by running `hal_i2c_slave_receive_dma()`. When a reception completes, `hal_i2c_slave_rx_cplt_callback()` will be called. You can overwrite the callback for certain operations.
5. If errors occur during data transmission/reception, `hal_i2c_error_callback()` will be called. You can overwrite the callback for certain operations.
6. Run `hal_i2c_master_abort_it()` to abort data transmission/reception as a master. If the abort completes, `hal_i2c_abort_cplt_callback()` will be called. You can overwrite the callback for certain operations.

2.16.2.6 I/O Memory Read and Write in DMA Mode

1. Write a large volume of data to a specified address in DMA mode by running `hal_i2c_mem_write_dma()`. When a write completes, `hal_i2c_mem_tx_cplt_callback()` will be called. You can overwrite the callback for certain operations.
2. Read a large volume of data from a specified address in DMA mode by running `hal_i2c_mem_read_dma()`. When a read completes, `hal_i2c_mem_rx_cplt_callback()` will be called. You can overwrite the callback for certain operations.
3. If errors occur during data transmission/reception, `hal_i2c_error_callback()` will be called. You can overwrite the callback for certain operations.

2.16.3 I2C Driver Structures

2.16.3.1 `i2c_init_t`

The initialization structure `i2c_init_t` of the I2C driver is defined below:

Table 2-194 `i2c_init_t` structure

| Data Field | Field Description | Value |
|---|---|--|
| <code>uint32_t speed</code> | Data transfer rate | This parameter can be one of the following values: <ul style="list-style-type: none"> • <code>I2C_SPEED_100K</code> (100 Kb/s) • <code>I2C_SPEED_400K</code> (400 Kb/s) • <code>I2C_SPEED_1000K</code> (1000 Kb/s) • <code>I2C_SPEED_2000K</code> (2.0 Mb/s) |
| <code>uint32_t own_address</code> | Local device address | 7-bit address: <code>0x08</code> to <code>0x77</code> 10-bit address: <code>0x008</code> to <code>0x077</code> , <code>0x080</code> to <code>0x3FE</code> |
| <code>uint32_t addressing_mode</code> | Format of local and peer device addresses | This parameter can be one of the following values: <ul style="list-style-type: none"> • <code>I2C_ADDRESSINGMODE_7BIT</code> (7-bit address) • <code>I2C_ADDRESSINGMODE_10BIT</code> (10-bit address) |
| <code>uint32_t general_call_mode</code> | Enable advertising address monitoring or not. | This parameter can be one of the following values: <ul style="list-style-type: none"> • <code>I2C_GENERALCALL_DISABLE</code> (disable) |

| Data Field | Field Description | Value |
|------------|-------------------|---|
| | | <ul style="list-style-type: none"> I2C_GENERALCALL_ENABLE (enable) |

2.16.3.2 i2c_handle_t

The handle structure `i2c_handle_t` of the I2C driver is defined below:

Table 2-195 `i2c_handle_t` structure

| Data Field | Field Description | Value |
|---|--|---|
| <code>i2c_regs_t *p_instance</code> | I2C peripheral instance | This parameter can be one of the following values: <ul style="list-style-type: none"> I2C0 I2C1 |
| <code>i2c_init_t init</code> | Initialization structure | See " Section 2.16.3.1 i2c_init_t ". |
| <code>uint8_t *p_buffer</code> | Pointer to data TX buffer (managed by I2C driver and initialization by developers not required) | N/A |
| <code>uint16_t xfer_size</code> | Data TX size (managed by I2C driver and initialization by developers not required) | N/A |
| <code>__IO uint16_t xfer_count</code> | Data TX count (managed by I2C driver and initialization by developers not required) | N/A |
| <code>__IO uint16_t master_ack_count</code> | ACK count for data reception as a master (managed by I2C driver and initialization by developers not required) | N/A |
| <code>__IO uint32_t xfer_options</code> | Sequential transfer option (managed by I2C driver and initialization by developers not required) | N/A |
| <code>__IO uint32_t previous_state</code> | Last communications status (managed by I2C driver and initialization by developers not required) | N/A |
| <code>hal_status_t(*xfer_isr)</code> (<code>struct_i2c_handle *p_i2c,</code> <code>uint32_t it_source, uint32_t</code> <code>abort_sources</code>) | Interrupt handler for data transfer (managed by I2C driver and initialization by developers not required) | N/A |
| <code>dma_handle_t *p_dmatx</code> | DMA handle pointer to I2C TX channel | Structure of DMA handle dma_handle_t for TX channels |
| <code>dma_handle_t *p_dmarx</code> | DMA handle pointer to I2C RX channel | Structure of DMA handle dma_handle_t for RX channels |
| <code>__IO hal_lock_t lock</code> | I2C lock (managed by I2C driver and initialization by developers not required) | N/A |

| Data Field | Field Description | Value |
|----------------------------|--|---|
| __IO hal_i2c_state_t state | I2C operating state (initialization by developers not required) | This parameter can be one of the following values: <ul style="list-style-type: none"> • HAL_I2C_STATE_RESET (not initialized) • HAL_I2C_STATE_READY (initialized and ready for use) • HAL_I2C_STATE_BUSY (busy) • HAL_I2C_STATE_BUSY_TX (TX ongoing) • HAL_I2C_STATE_BUSY_RX (RX ongoing) • HAL_I2C_STATE_ABORT (aborted) • HAL_I2C_STATE_TIMEOUT (timeout) • HAL_I2C_STATE_ERROR (error) |
| __IO hal_i2c_mode_t mode | I2C operating mode (managed by I2C driver and initialization by developers not required) | N/A |
| __IO uint32_t error_code | I2C error code (initialization by developers not required) | This parameter can be one of the following values: <ul style="list-style-type: none"> • HAL_I2C_ERROR_NONE (no error) • HAL_I2C_ERROR_ARB_LOST (arbitration loss) • HAL_I2C_ERROR_NOACK (no ACK) • HAL_I2C_ERROR_OVER (reception overflow) • HAL_I2C_ERROR_DMA (DMA transfer error) • HAL_I2C_ERROR_TIMEOUT (timeout) |
| uint32_t retention[10] | I2C register information (managed by I2C driver and initialization by developers not required) | N/A |

2.16.4 I2C Driver APIs

The I2C driver APIs are listed in the table below:

Table 2-196 I2C driver APIs

| API Type | API Name | Description |
|----------------|--------------------|---|
| Initialization | hal_i2c_init() | Initialize the I2C peripheral, and configure transfer rate and other parameters. |
| | hal_i2c_deinit() | Deinitialize the I2C peripheral. |
| | hal_i2c_msp_init() | Initialize GPIO pin multiplexing, NVIC interrupts, and DMA channels used by the I2C peripheral. |

| API Type | API Name | Description |
|---------------------------------------|--|---|
| | hal_i2c_msp_deinit() | Deinitialize GPIO pin multiplexing, NVIC interrupts, and DMA channels used by the I2C peripheral. |
| I/O operation | hal_i2c_master_transmit() | Transmit data as a master to a slave in polling mode. |
| | hal_i2c_master_receive() | Receive data as a master from a slave in polling mode. |
| | hal_i2c_slave_transmit() | Transmit data as a slave to a master in polling mode. |
| | hal_i2c_slave_receive() | Receive data as a slave from a master in polling mode. |
| | hal_i2c_mem_write() | Write data to a specified address in polling mode. |
| | hal_i2c_mem_read() | Read data from a specified address in polling mode. |
| | hal_i2c_master_transmit_it() | Transmit data as a master to a slave in interrupt mode. |
| | hal_i2c_master_receive_it() | Receive data as a master from a slave in interrupt mode. |
| | hal_i2c_slave_transmit_it() | Transmit data as a slave to a master in interrupt mode. |
| | hal_i2c_slave_receive_it() | Receive data as a slave from a master in interrupt mode. |
| | hal_i2c_mem_write_it() | Write data to a specified address in interrupt mode. |
| | hal_i2c_mem_read_it() | Read data from a specified address in interrupt mode. |
| | hal_i2c_master_sequential_transmit_it() | Transmit frame data as a master in interrupt mode. |
| | hal_i2c_master_sequential_receive_it() | Receive frame data as a master in interrupt mode. |
| | hal_i2c_slave_sequential_transmit_it() | Transmit frame data as a slave in interrupt mode. |
| hal_i2c_slave_sequential_receive_it() | Receive frame data as a slave in interrupt mode. | |
| hal_i2c_enable_listen_it() | Enable signal listening as a master in interrupt mode. | |

| API Type | API Name | Description |
|---------------------------------|-----------------------------------|---|
| | hal_i2c_disable_listen_it() | Disable signal listening as a master in interrupt mode. |
| | hal_i2c_master_transmit_dma() | Transmit data as a master to a slave in DMA mode. |
| | hal_i2c_master_receive_dma() | Receive data as a master from a slave in DMA mode. |
| | hal_i2c_slave_transmit_dma() | Transmit data as a slave to a master in DMA mode. |
| | hal_i2c_slave_receive_dma() | Receive data as a slave from a master in DMA mode. |
| | hal_i2c_mem_write_dma() | Write data to a specified address in DMA mode. |
| | hal_i2c_mem_read_dma() | Read data from a specified address in DMA mode. |
| | hal_i2c_master_abort_it() | Abort data transfer in interrupt/DMA mode. |
| Interrupt handling and callback | hal_i2c_irq_handler() | Interrupt handler |
| | hal_i2c_master_tx_cplt_callback() | TX complete interrupt callback for a master |
| | hal_i2c_master_rx_cplt_callback() | RX complete interrupt callback for a master |
| | hal_i2c_slave_tx_cplt_callback() | TX complete interrupt callback for a slave |
| | hal_i2c_slave_rx_cplt_callback() | RX complete interrupt callback for a slave |
| | hal_i2c_mem_tx_cplt_callback() | Write complete interrupt callback function |
| | hal_i2c_mem_rx_cplt_callback() | Read complete interrupt callback function |
| | hal_i2c_listen_cplt_callback() | Listening interrupt callback function |
| | hal_i2c_error_callback() | Error interrupt callback function |
| | hal_i2c_abort_cplt_callback() | Abort complete interrupt callback function |
| State and error | hal_i2c_get_state() | Get the driver operating state. |
| | hal_i2c_get_mode() | Get the current operating mode. |
| | hal_i2c_get_error() | Get error code. |
| Sleep | hal_i2c_suspend_reg() | Suspend registers related to I2C configuration in sleep mode. |
| | hal_i2c_resume_reg() | Resume registers related to I2C configuration during wakeup. |

The sections below elaborate on these APIs.

2.16.4.1 hal_i2c_init

Table 2-197 hal_i2c_init API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_i2c_init(i2c_handle_t *p_i2c) |
| Function Description | Initialize the I2C peripheral and related handles according to parameters of i2c_init_t . |
| Parameter | p_i2c: pointer to variables of i2c_handle_t . The variable contains the configuration information of a specified I2C. |
| Return Value | HAL status |
| Remarks | |

2.16.4.2 hal_i2c_deinit

Table 2-198 hal_i2c_deinit API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_i2c_deinit(i2c_handle_t *p_i2c) |
| Function Description | Deinitialize the I2C peripheral. |
| Parameter | p_i2c: pointer to variables of i2c_handle_t . The variable contains the configuration information of a specified I2C. |
| Return Value | HAL status |
| Remarks | |

2.16.4.3 hal_i2c_msp_init

Table 2-199 hal_i2c_msp_init API

| | |
|-----------------------------|---|
| Function Prototype | void hal_i2c_msp_init(i2c_handle_t *p_i2c) |
| Function Description | Initialize GPIO pin multiplexing, NVIC interrupts, and DMA channels used by I2C. |
| Parameter | p_i2c: pointer to variables of i2c_handle_t . The variable contains the configuration information of a specified I2C. |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite the API to initialize GPIO pin multiplexing, NVIC interrupts, and DMA channels. |

2.16.4.4 hal_i2c_msp_deinit

Table 2-200 hal_i2c_msp_deinit API

| | |
|-----------------------------|---|
| Function Prototype | void hal_i2c_msp_deinit(i2c_handle_t *p_i2c) |
| Function Description | Deinitialize GPIO pin multiplexing, NVIC interrupts, and DMA channels used by I2C. |
| Parameter | p_i2c: pointer to variables of i2c_handle_t . The variable contains the configuration information of a specified I2C. |

| | |
|---------------------|---|
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite the API to deinitialize GPIO pin multiplexing, NVIC interrupts, and DMA channels. |

2.16.4.5 hal_i2c_master_transmit

Table 2-201 hal_i2c_master_transmit API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_i2c_master_transmit(i2c_handle_t *p_i2c, uint16_t dev_address, uint8_t *p_data, uint16_t size, uint32_t timeout) |
| Function Description | Transmit a large volume of data as an I2C master in polling mode. |
| Parameter | p_i2c: pointer to variables of i2c_handle_t . The variable contains the configuration information of a specified I2C. dev_address: slave address p_data: pointer to data buffer size: size of data to be transmitted timeout: timeout period |
| Return Value | HAL status |
| Remarks | When it returns HAL_ERROR, you can call hal_i2c_get_error() to retrieve the error code. |

2.16.4.6 hal_i2c_master_receive

Table 2-202 hal_i2c_master_receive API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_i2c_master_receive(i2c_handle_t *p_i2c, uint16_t dev_address, uint8_t *p_data, uint16_t size, uint32_t timeout) |
| Function Description | Receive a large volume of data as an I2C master in polling mode. |
| Parameter | p_i2c: pointer to variables of i2c_handle_t . The variable contains the configuration information of a specified I2C. dev_address: slave address p_data: pointer to data buffer size: size of data to be received timeout: timeout period |
| Return Value | HAL status |
| Remarks | When it returns HAL_ERROR, you can call hal_i2c_get_error() to retrieve the error code. |

2.16.4.7 hal_i2c_slave_transmit

Table 2-203 hal_i2c_slave_transmit API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_i2c_slave_transmit(i2c_handle_t *p_i2c, uint8_t *p_data, uint16_t size, uint32_t timeout) |
| Function Description | Transmit a large volume of data as an I2C slave in polling mode. |
| Parameter | p_i2c: pointer to variables of i2c_handle_t . The variable contains the configuration information of a specified I2C. p_data: pointer to data buffer size: size of data to be transmitted timeout: timeout period |
| Return Value | HAL status |
| Remarks | When it returns HAL_ERROR, you can call hal_i2c_get_error() to retrieve the error code. |

2.16.4.8 hal_i2c_slave_receive

Table 2-204 hal_i2c_slave_receive API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_i2c_slave_receive(i2c_handle_t *p_i2c, uint8_t *p_data, uint16_t size, uint32_t timeout) |
| Function Description | Receive a large volume of data as an I2C slave in polling mode. |
| Parameter | p_i2c: pointer to variables of i2c_handle_t . The variable contains the configuration information of a specified I2C. p_data: pointer to data buffer size: size of data to be received timeout: timeout period |
| Return Value | HAL status |
| Remarks | When it returns HAL_ERROR, you can call hal_i2c_get_error() to retrieve the error code. |

2.16.4.9 hal_i2c_mem_write

Table 2-205 hal_i2c_mem_write API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_i2c_mem_write(i2c_handle_t *p_i2c, uint16_t dev_address, uint16_t mem_address, uint16_t mem_addr_size, uint8_t *p_data, uint16_t size, uint32_t timeout) |
| Function Description | Write a large volume of data to a specified slave address in polling mode. |
| Parameter | p_i2c: pointer to variables of i2c_handle_t . The variable contains the configuration information of a specified I2C. dev_address: slave address mem_address: specified internal slave address |

| | |
|---------------------|--|
| | <p>mem_addr_size: the specified internal slave address bit width. This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • I2C_MEMADD_SIZE_8BIT (8 bits) • I2C_MEMADD_SIZE_16BIT (16 bits) <p>p_data: pointer to data buffer</p> <p>size: size of data to be written</p> <p>timeout: timeout period</p> |
| Return Value | HAL status |
| Remarks | When it returns HAL_ERROR, you can call hal_i2c_get_error() to retrieve the error code. |

2.16.4.10 hal_i2c_mem_read

Table 2-206 hal_i2c_mem_read API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_i2c_mem_read(i2c_handle_t *p_i2c, uint16_t dev_address, uint16_t mem_address, uint16_t mem_addr_size, uint8_t *p_data, uint16_t size, uint32_t timeout) |
| Function Description | Read a large volume of data from a specified slave address in polling mode. |
| Parameter | <p>p_i2c: pointer to variables of i2c_handle_t. The variable contains the configuration information of a specified I2C.</p> <p>dev_address: slave address</p> <p>mem_address: specified internal slave address</p> <p>mem_addr_size: the specified internal slave address bit width. This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • I2C_MEMADD_SIZE_8BIT (8 bits) • I2C_MEMADD_SIZE_16BIT (16 bits) <p>p_data: pointer to data buffer</p> <p>size: size of data to be read</p> <p>timeout: timeout period</p> |
| Return Value | HAL status |
| Remarks | When it returns HAL_ERROR, you can call hal_i2c_get_error() to retrieve the error code. |

2.16.4.11 hal_i2c_master_transmit_it

Table 2-207 hal_i2c_master_transmit_it API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_i2c_master_transmit_it(i2c_handle_t *p_i2c, uint16_t dev_address, uint8_t *p_data, uint16_t size) |
| Function Description | Transmit a large amount of data as an I2C master in interrupt mode. |

| | |
|---------------------|---|
| Parameter | <p>p_i2c: pointer to variables of i2c_handle_t. The variable contains the configuration information of a specified I2C.</p> <p>dev_address: slave address</p> <p>p_data: pointer to data buffer</p> <p>size: size of data to be transmitted</p> |
| Return Value | HAL status |
| Remarks | <ul style="list-style-type: none"> When a transmission completes, the callback function hal_i2c_master_tx_cplt_callback() will be called. When an error occurs during transmission, the callback function hal_i2c_error_callback() will be called. You can call hal_i2c_get_error() when running the callback function to retrieve the related error code. Before calling hal_i2c_master_tx_cplt_callback(), do not release the memory of the data buffer pointed by data. During transmission, if the I2C interrupt handler cannot respond in time, transmission may fail. |

2.16.4.12 hal_i2c_master_receive_it

Table 2-208 hal_i2c_master_receive_it API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_i2c_master_receive_it(i2c_handle_t *p_i2c, uint16_t dev_address, uint8_t *p_data, uint16_t size) |
| Function Description | Receive a large amount of data as an I2C master in interrupt mode. |
| Parameter | <p>p_i2c: pointer to variables of i2c_handle_t. The variable contains the configuration information of a specified I2C.</p> <p>dev_address: slave address</p> <p>p_data: pointer to data buffer</p> <p>size: size of data to be received</p> |
| Return Value | HAL status |
| Remarks | <ul style="list-style-type: none"> When a reception completes, the callback function hal_i2c_master_rx_cplt_callback() will be called. When an error occurs during reception, the callback function hal_i2c_error_callback() will be called. You can call hal_i2c_get_error() when running the callback function to retrieve the related error code. Before calling hal_i2c_master_rx_cplt_callback(), do not release the memory of the data buffer pointed by data. During transmission, if the I2C interrupt handler cannot respond in time, reception may fail. |

2.16.4.13 hal_i2c_slave_transmit_it

Table 2-209 hal_i2c_slave_transmit_it API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_i2c_slave_transmit_it(i2c_handle_t *p_i2c, uint8_t *p_data, uint16_t size) |
| Function Description | Transmit a large amount of data as an I2C slave in interrupt mode. |
| Parameter | <p>p_i2c: pointer to variables of i2c_handle_t. The variable contains the configuration information of a specified I2C.</p> <p>p_data: pointer to data buffer</p> <p>size: size of data to be transmitted</p> |
| Return Value | HAL status |
| Remarks | <ul style="list-style-type: none"> • When a transmission completes, the callback function hal_i2c_slave_tx_cplt_callback() will be called. • When an error occurs during transmission, the callback function hal_i2c_error_callback() will be called. You can call hal_i2c_get_error() when running the callback function to retrieve the related error code. • Before calling hal_i2c_slave_tx_cplt_callback(), do not release the memory of the data buffer pointed by data. • During transmission, if the I2C interrupt handler cannot respond in time, a data transmission error may occur. |

2.16.4.14 hal_i2c_slave_receive_it

Table 2-210 hal_i2c_slave_receive_it API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_i2c_slave_receive_it(i2c_handle_t *p_i2c, uint8_t *p_data, uint16_t size) |
| Function Description | Receive a large amount of data as an I2C slave in interrupt mode. |
| Parameter | <p>p_i2c: pointer to variables of i2c_handle_t. The variable contains the configuration information of a specified I2C.</p> <p>p_data: pointer to data buffer</p> <p>size: size of data to be received</p> |
| Return Value | HAL status |
| Remarks | <ul style="list-style-type: none"> • When a reception completes, the callback function hal_i2c_slave_rx_cplt_callback() will be called. • When an error occurs during reception, the callback function hal_i2c_error_callback() will be called. You can call hal_i2c_get_error() when running the callback function to retrieve the related error code. • Before calling hal_i2c_slave_rx_cplt_callback(), do not release the memory of the data buffer pointed by data. • During transmission, if the I2C interrupt handler cannot respond in time, a data reception error may occur. |

2.16.4.15 hal_i2c_mem_write_it

Table 2-211 hal_i2c_mem_write_it API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_i2c_mem_write_it(i2c_handle_t *p_i2c, uint16_t dev_address, uint16_t mem_address, uint16_t mem_addr_size, uint8_t *p_data, uint16_t size) |
| Function Description | Write a large volume of data to a specified slave address in interrupt mode. |
| Parameter | <p>p_i2c: pointer to variables of i2c_handle_t. The variable contains the configuration information of a specified I2C.</p> <p>dev_address: slave address</p> <p>mem_address: specified internal slave address</p> <p>mem_addr_size: the specified internal slave address bit width. This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • I2C_MEMADD_SIZE_8BIT (8 bits) • I2C_MEMADD_SIZE_16BIT (16 bits) <p>p_data: pointer to data buffer</p> <p>size: size of data to be written</p> |
| Return Value | HAL status |
| Remarks | <ul style="list-style-type: none"> • When a write completes, the callback function hal_i2c_mem_tx_cplt_callback() will be called. • When an error occurs during write, the callback function hal_i2c_error_callback() will be called. You can call hal_i2c_get_error() when running the callback function to retrieve the related error code. • Before calling hal_i2c_mem_tx_cplt_callback(), do not release the memory of the data buffer pointed by data. • During transmission, if the I2C interrupt handler cannot respond in time, transmission may fail. |

2.16.4.16 hal_i2c_mem_read_it

Table 2-212 hal_i2c_mem_read_it API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_i2c_mem_read_it(i2c_handle_t *p_i2c, uint16_t dev_address, uint16_t mem_address, uint16_t mem_addr_size, uint8_t *p_data, uint16_t size) |
| Function Description | Read a large volume of data from a specified slave address in interrupt mode. |
| Parameter | <p>p_i2c: pointer to variables of i2c_handle_t. The variable contains the configuration information of a specified I2C.</p> <p>dev_address: slave address</p> <p>mem_address: specified internal slave address</p> <p>mem_addr_size: the specified internal slave address bit width. This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • I2C_MEMADD_SIZE_8BIT (8 bits) |

| | |
|---------------------|--|
| | <ul style="list-style-type: none"> I2C_MEMADD_SIZE_16BIT (16 bits) <p>p_data: pointer to data buffer</p> <p>size: size of data to be read</p> |
| Return Value | HAL status |
| Remarks | <ul style="list-style-type: none"> When a read completes, the callback function hal_i2c_mem_rx_cplt_callback() will be called. When an error occurs during read, the callback function hal_i2c_error_callback() will be called. You can call hal_i2c_get_error() when running the callback function to retrieve the related error code. Before calling hal_i2c_mem_rx_cplt_callback(), do not release the memory of the data buffer pointed by data. During transmission, if the I2C interrupt handler cannot respond in time, reception may fail. |

2.16.4.17 hal_i2c_master_abort_it

Table 2-213 hal_i2c_master_abort_it API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_i2c_master_abort_it(i2c_handle_t *p_i2c) |
| Function Description | In interrupt mode, abort data transfer from an I2C master in interrupt/DMA mode. |
| Parameter | p_i2c: pointer to variables of i2c_handle_t . The variable contains the configuration information of a specified I2C. |
| Return Value | HAL status |
| Remarks | This is a non-polling function, and returns status immediately after enabling TX_ABRT interrupt. When an abort completes after TX_ABRT interrupt is triggered, hal_i2c_abort_cplt_callback() is called. |

2.16.4.18 hal_i2c_master_transmit_dma

Table 2-214 hal_i2c_master_transmit_dma API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_i2c_master_transmit_dma(i2c_handle_t *p_i2c, uint16_t dev_address, uint8_t *p_data, uint16_t size) |
| Function Description | Transmit a large volume of data as an I2C master in DMA mode. |
| Parameter | <p>p_i2c: pointer to variables of i2c_handle_t. The variable contains the configuration information of a specified I2C.</p> <p>dev_address: slave address</p> <p>p_data: pointer to data buffer</p> <p>size: size of data to be transmitted</p> |
| Return Value | HAL status |
| Remarks | <ul style="list-style-type: none"> When a transmission completes, the callback function hal_i2c_master_tx_cplt_callback() will be called. |

| | |
|--|---|
| | <ul style="list-style-type: none"> When an error occurs during transmission, the callback function hal_i2c_error_callback() will be called. You can call hal_i2c_get_error() when running the callback function to retrieve the related error code. Before calling hal_i2c_master_tx_cplt_callback(), do not release the memory of the data buffer pointed by data. During transmission, if the I2C interrupt handler cannot respond in time, transmission may fail. |
|--|---|

2.16.4.19 hal_i2c_master_receive_dma

Table 2-215 hal_i2c_master_receive_dma API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_i2c_master_receive_dma(i2c_handle_t *p_i2c, uint16_t dev_address, uint8_t *p_data, uint16_t size) |
| Function Description | Receive a large amount of data as an I2C master in DMA mode. |
| Parameter | <p>p_i2c: pointer to variables of i2c_handle_t. The variable contains the configuration information of a specified I2C.</p> <p>dev_address: slave address</p> <p>p_data: pointer to data buffer</p> <p>size: size of data to be received</p> |
| Return Value | HAL status |
| Remarks | <ul style="list-style-type: none"> When a reception completes, the callback function hal_i2c_master_rx_cplt_callback() will be called. When an error occurs during reception, the callback function hal_i2c_error_callback() will be called. You can call hal_i2c_get_error() when running the callback function to retrieve the related error code. Before calling hal_i2c_master_rx_cplt_callback(), do not release the memory of the data buffer pointed by data. |

2.16.4.20 hal_i2c_slave_transmit_dma

Table 2-216 hal_i2c_slave_transmit_dma API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_i2c_slave_transmit_dma(i2c_handle_t *p_i2c, uint8_t *p_data, uint16_t size) |
| Function Description | Transmit a large amount of data as an I2C slave in DMA mode. |
| Parameter | <p>p_i2c: pointer to variables of i2c_handle_t. The variable contains the configuration information of a specified I2C.</p> <p>p_data: pointer to data buffer</p> <p>size: size of data to be transmitted</p> |
| Return Value | HAL status |
| Remarks | <ul style="list-style-type: none"> When a transmission completes, the callback function hal_i2c_slave_tx_cplt_callback() will be called. |

- When an error occurs during transmission, the callback function [hal_i2c_error_callback\(\)](#) will be called. You can call [hal_i2c_get_error\(\)](#) when running the callback function to retrieve the related error code.
- Before calling [hal_i2c_slave_tx_cplt_callback\(\)](#), do not release the memory of the data buffer pointed by data.

2.16.4.21 hal_i2c_slave_receive_dma

Table 2-217 hal_i2c_slave_receive_dma API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_i2c_slave_receive_dma(i2c_handle_t *p_i2c, uint8_t *p_data, uint16_t size) |
| Function Description | Receive a large amount of data as an I2C slave in DMA mode. |
| Parameter | <p>p_i2c: pointer to variables of i2c_handle_t. The variable contains the configuration information of a specified I2C.</p> <p>p_data: pointer to data buffer</p> <p>size: size of data to be received</p> |
| Return Value | HAL status |
| Remarks | <ul style="list-style-type: none"> • When a reception completes, the callback function hal_i2c_slave_rx_cplt_callback() will be called. • When an error occurs during reception, the callback function hal_i2c_error_callback() will be called. You can call hal_i2c_get_error() when running the callback function to retrieve the related error code. • Before calling hal_i2c_slave_rx_cplt_callback(), do not release the memory of the data buffer pointed by data. |

2.16.4.22 hal_i2c_mem_write_dma

Table 2-218 hal_i2c_mem_write_dma API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_i2c_mem_write_dma(i2c_handle_t *p_i2c, uint16_t dev_address, uint16_t mem_address, uint16_t mem_addr_size, uint8_t *p_data, uint16_t size) |
| Function Description | Write a large volume of data to a specified slave address in DMA mode. |
| Parameter | <p>p_i2c: pointer to variables of i2c_handle_t. The variable contains the configuration information of a specified I2C.</p> <p>dev_address: slave address</p> <p>mem_address: specified internal slave address</p> <p>mem_addr_size: the specified internal slave address bit width. This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • I2C_MEMADD_SIZE_8BIT (8 bits) • I2C_MEMADD_SIZE_16BIT (16 bits) |

| | |
|---------------------|---|
| | <p>p_data: pointer to data buffer</p> <p>size: size of data to be written</p> |
| Return Value | HAL status |
| Remarks | <ul style="list-style-type: none"> When a write completes, the callback function hal_i2c_mem_tx_cplt_callback() will be called. When an error occurs during write, the callback function hal_i2c_error_callback() will be called. You can call hal_i2c_get_error() when running the callback function to retrieve the related error code. Before calling hal_i2c_mem_tx_cplt_callback(), do not release the memory of the data buffer pointed by data. |

2.16.4.23 hal_i2c_mem_read_dma

Table 2-219 hal_i2c_mem_read_dma API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_i2c_mem_read_dma(i2c_handle_t *p_i2c, uint16_t dev_address, uint16_t mem_address, uint16_t mem_addr_size, uint8_t *p_data, uint16_t size) |
| Function Description | Read a large volume of data from a specified slave address in DMA mode. |
| Parameter | <p>p_i2c: pointer to variables of i2c_handle_t. The variable contains the configuration information of a specified I2C.</p> <p>dev_address: slave address</p> <p>mem_address: specified internal slave address</p> <p>mem_addr_size: the specified internal slave address bit width. This parameter can be one of the following values:</p> <ul style="list-style-type: none"> I2C_MEMADD_SIZE_8BIT (8 bits) I2C_MEMADD_SIZE_16BIT (16 bits) <p>p_data: pointer to data buffer</p> <p>size: size of data to be read</p> |
| Return Value | HAL status |
| Remarks | <ul style="list-style-type: none"> When a read completes, the callback function hal_i2c_mem_rx_cplt_callback() will be called. When an error occurs during read, the callback function hal_i2c_error_callback() will be called. You can call hal_i2c_get_error() when running the callback function to retrieve the related error code. Before calling hal_i2c_mem_rx_cplt_callback(), do not release the memory of the data buffer pointed by data. |

2.16.4.24 hal_i2c_irq_handler

Table 2-220 hal_i2c_irq_handler API

| | |
|---------------------------|---|
| Function Prototype | void hal_i2c_irq_handler(i2c_handle_t *p_i2c) |
|---------------------------|---|

| | |
|-----------------------------|---|
| Function Description | Handle I2C interrupt requests. |
| Parameter | p_i2c: pointer to variables of i2c_handle_t . The variable contains the configuration information of a specified I2C. |
| Return Value | None |
| Remarks | |

2.16.4.25 hal_i2c_master_tx_cplt_callback

Table 2-221 hal_i2c_master_tx_cplt_callback API

| | |
|-----------------------------|---|
| Function Prototype | void hal_i2c_master_tx_cplt_callback(i2c_handle_t *p_i2c) |
| Function Description | Transmission complete callback function for a master |
| Parameter | p_i2c: pointer to variables of i2c_handle_t . The variable contains the configuration information of a specified I2C. |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite this function before using it. |

2.16.4.26 hal_i2c_master_rx_cplt_callback

Table 2-222 hal_i2c_master_rx_cplt_callback API

| | |
|-----------------------------|---|
| Function Prototype | void hal_i2c_master_rx_cplt_callback(i2c_handle_t *p_i2c) |
| Function Description | Reception complete callback function for a master |
| Parameter | p_i2c: pointer to variables of i2c_handle_t . The variable contains the configuration information of a specified I2C. |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite this function before using it. |

2.16.4.27 hal_i2c_slave_tx_cplt_callback

Table 2-223 hal_i2c_slave_tx_cplt_callback API

| | |
|-----------------------------|---|
| Function Prototype | void hal_i2c_slave_tx_cplt_callback(i2c_handle_t *p_i2c) |
| Function Description | Transmission complete callback function for a slave |
| Parameter | p_i2c: pointer to variables of i2c_handle_t . The variable contains the configuration information of a specified I2C. |
| Return Value | None |

| | |
|----------------|--|
| Remarks | The function is declared empty as weak function. Developers are required to overwrite this function before using it. |
|----------------|--|

2.16.4.28 hal_i2c_slave_rx_cplt_callback

Table 2-224 hal_i2c_slave_rx_cplt_callback API

| | |
|-----------------------------|---|
| Function Prototype | void hal_i2c_slave_rx_cplt_callback(i2c_handle_t *p_i2c) |
| Function Description | Reception complete callback function for a slave |
| Parameter | p_i2c: pointer to variables of i2c_handle_t . The variable contains the configuration information of a specified I2C. |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite this function before using it. |

2.16.4.29 hal_i2c_mem_tx_cplt_callback

Table 2-225 hal_i2c_mem_tx_cplt_callback API

| | |
|-----------------------------|---|
| Function Prototype | void hal_i2c_mem_tx_cplt_callback(i2c_handle_t *p_i2c) |
| Function Description | Write complete callback function for a slave |
| Parameter | p_i2c: pointer to variables of i2c_handle_t . The variable contains the configuration information of a specified I2C. |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite this function before using it. |

2.16.4.30 hal_i2c_mem_rx_cplt_callback

Table 2-226 hal_i2c_mem_rx_cplt_callback API

| | |
|-----------------------------|---|
| Function Prototype | void hal_i2c_mem_rx_cplt_callback(i2c_handle_t *p_i2c) |
| Function Description | Read complete callback function for a slave |
| Parameter | p_i2c: pointer to variables of i2c_handle_t . The variable contains the configuration information of a specified I2C. |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite this function before using it. |

2.16.4.31 hal_i2c_error_callback

Table 2-227 hal_i2c_error_callback API

| | |
|-----------------------------|---|
| Function Prototype | void hal_i2c_error_callback(i2c_handle_t *p_i2c) |
| Function Description | I2C error callback function |
| Parameter | p_i2c: pointer to variables of i2c_handle_t . The variable contains the configuration information of a specified I2C. |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite this function before using it. |

2.16.4.32 hal_i2c_abort_cplt_callback

Table 2-228 hal_i2c_abort_cplt_callback API

| | |
|-----------------------------|---|
| Function Prototype | void hal_i2c_abort_cplt_callback(i2c_handle_t *p_i2c) |
| Function Description | I2C abort complete callback function |
| Parameter | p_i2c: pointer to variables of i2c_handle_t . The variable contains the configuration information of a specified I2C. |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite this function before using it. |

2.16.4.33 hal_i2c_get_state

Table 2-229 hal_i2c_get_state API

| | |
|-----------------------------|---|
| Function Prototype | hal_i2c_state_t hal_i2c_get_state(i2c_handle_t *p_i2c) |
| Function Description | Get the I2C operating state. |
| Parameter | p_i2c: pointer to variables of i2c_handle_t . The variable contains the configuration information of a specified I2C. |
| Return Value | I2C operating state. This parameter can be one of the following values: <ul style="list-style-type: none"> • HAL_I2C_STATE_RESET (not initialized) • HAL_I2C_STATE_READY (initialized and ready for use) • HAL_I2C_STATE_BUSY (busy) • HAL_I2C_STATE_BUSY_TX (TX ongoing) • HAL_I2C_STATE_BUSY_RX (RX ongoing) • HAL_I2C_STATE_LISTEN (address listening ongoing) • HAL_I2C_STATE_BUSY_TX_LISTEN (TX and address listening ongoing) • HAL_I2C_STATE_BUSY_RX_LISTEN (RX and address listening ongoing) |

| | |
|---------|---|
| | <ul style="list-style-type: none"> • HAL_I2C_STATE_ABORT (aborted) • HAL_I2C_STATE_TIMEOUT (timeout) • HAL_I2C_STATE_ERROR (error) |
| Remarks | |

2.16.4.34 hal_i2c_get_mode

Table 2-230 hal_i2c_get_mode API

| | |
|----------------------|--|
| Function Prototype | hal_i2c_mode_t hal_i2c_get_mode(i2c_handle_t *p_i2c) |
| Function Description | Return I2C mode: master, slave, memory, or none |
| Parameter | p_i2c: pointer to variables of i2c_handle_t . The variable contains the configuration information of a specified I2C. |
| Return Value | <p>I2C mode. This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • HAL_I2C_MODE_NONE (none) • HAL_I2C_MODE_MASTER (master) • HAL_I2C_MODE_SLAVE (slave) • HAL_I2C_MODE_MEM (read and write memory) |
| Remarks | |

2.16.4.35 hal_i2c_get_error

Table 2-231 hal_i2c_get_error API

| | |
|----------------------|--|
| Function Prototype | uint32_t hal_i2c_get_error(i2c_handle_t *p_i2c) |
| Function Description | Return the I2C handle error code. |
| Parameter | p_i2c: pointer to variables of i2c_handle_t . The variable contains the configuration information of a specified I2C. |
| Return Value | <p>I2C error code. This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • HAL_I2C_ERROR_NONE (no error) • HAL_I2C_ERROR_ARB_LOST (arbitration loss) • HAL_I2C_ERROR_NOACK (no ACK) • HAL_I2C_ERROR_OVER (reception overflow) • HAL_I2C_ERROR_DMA (DMA transfer error) • HAL_I2C_ERROR_TIMEOUT (timeout) |
| Remarks | |

2.16.4.36 hal_i2c_suspend_reg

Table 2-232 hal_i2c_suspend_reg API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_i2c_suspend_reg(i2c_handle_t *p_i2c) |
| Function Description | Suspend registers related to I2C configuration in sleep mode. |
| Parameter | p_i2c: pointer to variables of i2c_handle_t . The variable contains the configuration information of a specified I2C. |
| Return Value | HAL status |
| Remarks | |

2.16.4.37 hal_i2c_resume_reg

Table 2-233 hal_i2c_resume_reg API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_i2c_resume_reg(i2c_handle_t *p_i2c) |
| Function Description | Resume registers related to I2C configuration during wakeup. |
| Parameter | p_i2c: pointer to variables of i2c_handle_t . The variable contains the configuration information of a specified I2C. |
| Return Value | HAL status |
| Remarks | |

2.17 HAL QSPI Generic Driver

2.17.1 QSPI Driver Functionalities

The HAL Quad-SPI (QSPI) driver features the following functionalities:

- Three data transfer modes: Standard, Dual, and Quad.
- Up to 32 bits wide for data transfer
- Transfer rate at up to 32 MHz (in Standard mode)
- Configurable clock polarity (CPOL) and clock phase (CPHA)
- Configurable size and transmission mode for commands and addresses
- Setting and obtaining TX/RX FIFO thresholds
- Three data read and write approaches: polling, interrupt, and DMA
- Aborting data read and write in interrupt/DMA mode
- TX/RX complete, error, and abort complete interrupt callback functions
- Getting operating state and error code of QSPI driver
- Timeout settings

2.17.2 How to Use QSPI Driver

Developers can use the QSPI driver in the following scenarios:

1. Declare a `qspi_handle_t` handle structure variable, for example: `qspi_handle_t qspi_handle`.
2. Initialize the QSPI low-level resources by overwriting `hal_qspi_msp_init()`:
 - (1). QSPI pin configuration: Configure the GPIO mode as `GPIO_MODE_MUX` (multiplexing mode) by calling `hal_gpio_init()`, and configure the multiplexing functionalities of relevant GPIOs as QSPI.
 - (2). To use the interrupt process (`hal_qspi_transmit_it()` and `hal_qspi_receive_it()` APIs), developers need to call related NVIC APIs:
 - Configure the QSPI interrupt priority by calling `hal_nvic_set_priority()`.
 - Enable QSPI interrupt handling by calling `hal_nvic_enable_irq()`.
 - (3). If you need to use DMA process (`hal_qspi_transmit_dma()` and `hal_qspi_receive_dma()` APIs), you need to configure DMA:
 - Declare a DMA channel for TX/RX channels.
 - Declare a DMA handle structure for TX/RX channels, for example: `dma_handle_t hdma`.
 - Configure the declared DMA handle structure by using the required TX/RX parameters.
 - Configure DMA TX/RX channels.
 - Associate the initialized DMA handle with QSPI DMA TX/RX handles.
 - Configure the priority and enable the NVIC for transfer complete interrupt on DMA TX/RX channels.
 - (4). Configure parameters, such as clock prescaler values, in the init structure in the `qspi_handle` handle.
 - (5). Initialize QSPI registers by calling `hal_qspi_init()`.

2.17.3 QSPI Driver Structures

2.17.3.1 qspi_init_t

The initialization structure `qspi_init_t` of the QSPI driver is defined below:

Table 2-234 `qspi_init_t` structure

| Data Field | Field Description | Value |
|---------------------------------------|-------------------------------|--|
| <code>uint32_t clock_prescaler</code> | Clock prescaler value | Even numbers between 0x0000 and 0xFFFF. QSPI transfer rate = system clock / prescaler value |
| <code>uint32_t clock_mode</code> | Clock polarity and phase mode | This parameter can be one of the following values: <ul style="list-style-type: none"> • <code>QSPI_CLOCK_MODE_0</code> (CPOL = 0, CPHA = 0) • <code>QSPI_CLOCK_MODE_1</code> (CPOL = 0, CPHA = 1) • <code>QSPI_CLOCK_MODE_2</code> (CPOL = 1, CPHA = 0) |

| Data Field | Field Description | Value |
|--------------------------|------------------------|---|
| | | <ul style="list-style-type: none"> QSPI_CLOCK_MODE_3 (CPOL =1, CPHA = 1) |
| uint32_t rx_sample_delay | RX delayed acquisition | 0x0 to 0x7 |

2.17.3.2 qspi_handle_t

The handle structure `qspi_handle_t` of the QSPI driver is defined below:

Table 2-235 `qspi_handle_t` structure

| Data Field | Field Description | Value |
|---|--|---|
| <code>ssi_regs_t*p_instance</code> | QSPI peripheral instance | This parameter can be one of the following values: <ul style="list-style-type: none"> QSPI0 QSPI1 |
| <code>qspi_init_t init</code> | Initialization structure (see " Section 2.17.3.1 qspi_init_t ".) | N/A |
| <code>uint8_t *p_tx_buffer</code> | Pointer to data TX buffer (managed by QSPI driver and initialization by developers not required) | N/A |
| <code>__IO uint32_t tx_buffer_size</code> | Data TX size (managed by QSPI driver and initialization by developers not required) | N/A |
| <code>__IO uint32_t tx_xfer_count</code> | Data TX count (managed by QSPI driver and initialization by developers not required) | N/A |
| <code>uint8_t *p_rx_buffer</code> | Pointer to data RX buffer (managed by QSPI driver and initialization by developers not required) | N/A |
| <code>__IO uint32_t rx_buffer_size</code> | Data RX size (managed by QSPI driver and initialization by developers not required) | N/A |
| <code>__IO uint32_t rx_xfer_count</code> | Data RX count (managed by QSPI driver and initialization by developers not required) | N/A |
| <code>void (*write_fifo)(struct_qspi_handle *p_qspi)</code> | Pointer to FIFO functions written during QSPI TX (managed by QSPI | N/A |

| Data Field | Field Description | Value |
|---|--|--|
| | driver and initialization by developers not required) | |
| void (*read_fifo)(struct_qspi_handle *p_qspi) | Pointer to FIFO functions read during QSPI RX (managed by QSPI driver and initialization by developers not required) | N/A |
| dma_handle_t *p_dma | Pointer to dma_handle_t structure of DMA handle for data TX/RX channels | N/A |
| __IO hal_lock_t lock | QSPI lock (managed by QSPI driver and initialization by developers not required) | N/A |
| __IO hal_qspi_state_t state | QSPI operating state (initialization by developers not required) | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • HAL_QSPI_STATE_RESET (not initialized) • HAL_QSPI_STATE_READY (initialized and ready for use) • HAL_QSPI_STATE_BUSY (busy) • HAL_QSPI_STATE_BUSY_INDIRECT_TX (TX ongoing) • HAL_QSPI_STATE_BUSY_INDIRECT_RX (RX ongoing) • HAL_QSPI_STATE_ABORT (aborted) • HAL_QSPI_STATE_ERROR (error) |
| __IO uint32_t error_code | QSPI error code (initialization by developers not required) | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • HAL_QSPI_ERROR_NONE (no error) • HAL_QSPI_ERROR_TIMEOUT (timeout) • HAL_QSPI_ERROR_TRANSFER (transfer error) • HAL_QSPI_ERROR_DMA (DMA transfer error) • HAL_QSPI_ERROR_INVALID_PARAM (invalid parameter) |
| uint32_t timeout | QSPI timeout period (initialization by developers not required) | N/A |

| Data Field | Field Description | Value |
|-----------------------|--|-------|
| uint32_t retention[8] | QSPI register information (managed by QSPI driver and initialization by developers not required) | N/A |

2.17.3.3 qspi_command_t

The command structure `qspi_command_t` of the QSPI driver is defined below:

Table 2-236 `qspi_command_t` structure

| Data Field | Field Description | Value |
|---------------------------|--|--|
| uint32_t instruction | Instruction | 0x0000 to 0xFFFF |
| uint32_t address | Address | 0x0000_0000 to 0xFFFF_FFFF |
| uint32_t instruction_size | Instruction size | This parameter can be one of the following values: <ul style="list-style-type: none"> • QSPI_INSTSIZE_00_BITS (0 bit) • QSPI_INSTSIZE_04_BITS (4 bits) • QSPI_INSTSIZE_08_BITS (8 bits) • QSPI_INSTSIZE_16_BITS (16 bits) |
| uint32_t address_size | Address bit width | This parameter can be one of the following values: <ul style="list-style-type: none"> • QSPI_ADDRSIZE_00_BITS (0 bit) • QSPI_ADDRSIZE_04_BITS (4 bits) • QSPI_ADDRSIZE_08_BITS (8 bits) • QSPI_ADDRSIZE_12_BITS (12 bits) • QSPI_ADDRSIZE_16_BITS (16 bits) • QSPI_ADDRSIZE_20_BITS (20 bits) • QSPI_ADDRSIZE_24_BITS (24 bits) • QSPI_ADDRSIZE_28_BITS (28 bits) • QSPI_ADDRSIZE_32_BITS (32 bits) |
| uint32_t dummy_cycles | Clock cycle inserted during read and write switching | 0 to 31 |
| uint32_t data_size | Valid data bit | This parameter can be one of the following values: <ul style="list-style-type: none"> • LL_SSI_DATASIZE_4BIT • LL_SSI_DATASIZE_5BIT • LL_SSI_DATASIZE_6BIT • LL_SSI_DATASIZE_7BIT |

| Data Field | Field Description | Value |
|-----------------------------------|--|---|
| | | <ul style="list-style-type: none"> • LL_SSI_DATASIZE_8BIT • LL_SSI_DATASIZE_9BIT • LL_SSI_DATASIZE_10BIT • LL_SSI_DATASIZE_11BIT • LL_SSI_DATASIZE_12BIT • LL_SSI_DATASIZE_13BIT • LL_SSI_DATASIZE_14BIT • LL_SSI_DATASIZE_15BIT • LL_SSI_DATASIZE_16BIT • LL_SSI_DATASIZE_17BIT • LL_SSI_DATASIZE_18BIT • LL_SSI_DATASIZE_19BIT • LL_SSI_DATASIZE_20BIT • LL_SSI_DATASIZE_21BIT • LL_SSI_DATASIZE_22BIT • LL_SSI_DATASIZE_23BIT • LL_SSI_DATASIZE_24BIT • LL_SSI_DATASIZE_25BIT • LL_SSI_DATASIZE_26BIT • LL_SSI_DATASIZE_27BIT • LL_SSI_DATASIZE_28BIT • LL_SSI_DATASIZE_29BIT • LL_SSI_DATASIZE_30BIT • LL_SSI_DATASIZE_31BIT • LL_SSI_DATASIZE_32BIT |
| uint32_t instruction_address_mode | Transfer mode for instructions and addresses | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • QSPI_INST_ADDR_ALL_IN_SPI (transfer instructions and addresses in Standard SPI mode) • QSPI_INST_IN_SPI_ADDR_IN_SPIFRF (transfer instructions in Standard SPI mode; transfer addresses in Dual/Quad SPI mode) • QSPI_INST_ADDR_ALL_IN_SPIFRF (transfer instructions and addresses in Dual/Quad SPI mode) |
| uint32_t data_mode | Data transfer mode | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • QSPI_DATA_MODE_SPI (in Standard SPI mode) |

| Data Field | Field Description | Value |
|-----------------|-------------------|--|
| | | <ul style="list-style-type: none"> QSPI_DATA_MODE_DUALSPI (in Dual SPI mode) QSPI_DATA_MODE_QUADSPI (in Quad SPI mode) |
| uint32_t length | Data size | 0x0000_0000 to 0xFFFF_FFFF |

2.17.4 QSPI Driver APIs

The QSPI driver APIs are listed in the table below:

Table 2-237 QSPI driver APIs

| API Type | API Name | Description |
|------------------------|------------------------------------|--|
| Initialization | hal_qspi_init() | Initialize the QSPI peripheral, and configure clock prescaler values and other parameters. |
| | hal_qspi_deinit() | Deinitialize the QSPI peripheral. |
| | hal_qspi_msp_init() | Initialize GPIO pin multiplexing, NVIC interrupts, and DMA channels used by the QSPI peripheral. |
| | hal_qspi_msp_deinit() | Deinitialize GPIO pin multiplexing, NVIC interrupts, and DMA channels used by the QSPI peripheral. |
| I/O operation | hal_qspi_command_transmit() | Transmit data (including instructions, addresses, and data) in polling mode. |
| | hal_qspi_command_receive() | Receive data (including instructions, addresses, and data) in polling mode. |
| | hal_qspi_command() | Transmit instructions in polling mode. |
| | hal_qspi_transmit() | Transmit data (in SPI mode) in polling mode. |
| | hal_qspi_receive() | Receive data (in SPI mode) in polling mode. |
| | hal_qspi_command_transmit_it() | Transmit data (including instructions, addresses, and data) in interrupt mode. |
| | hal_qspi_command_receive_it() | Receive data (including instructions, addresses, and data) in interrupt mode. |
| | hal_qspi_command_it() | Transmit instructions in interrupt mode. |
| | hal_qspi_transmit_it() | Transmit data (in SPI mode) in interrupt mode. |
| | hal_qspi_receive_it() | Receive data (in SPI mode) in interrupt mode. |
| | hal_qspi_command_transmit_dma() | Transmit data (including instructions, addresses, and data) in DMA mode. |
| | hal_qspi_command_receive_dma() | Receive data (including instructions, addresses, and data) in DMA mode. |
| hal_qspi_command_dma() | Transmit instructions in DMA mode. | |

| API Type | API Name | Description |
|---------------------------------|----------------------------------|--|
| | hal_qspi_transmit_dma() | Transmit data (in SPI mode) in DMA mode. |
| | hal_qspi_receive_dma() | Receive data (in SPI mode) in DMA mode. |
| | hal_qspi_abort() | In polling mode, abort data transfer in interrupt/DMA mode. |
| | hal_qspi_abort_it() | In interrupt mode, abort data transfer in interrupt/DMA mode. |
| Interrupt handling and callback | hal_qspi_irq_handler() | Interrupt handler |
| | hal_qspi_tx_cplt_callback() | TX complete interrupt callback |
| | hal_qspi_rx_cplt_callback() | RX complete interrupt callback |
| | hal_qspi_error_callback() | Error interrupt callback function |
| | hal_qspi_abort_cplt_callback() | Abort complete interrupt callback function |
| State and error | hal_qspi_get_state() | Get the driver operating state. |
| | hal_qspi_get_error() | Get error code. |
| Control | hal_qspi_set_timeout() | Set a timeout period. |
| | hal_qspi_set_tx_fifo_threshold() | Set a TX FIFO threshold. |
| | hal_qspi_set_rx_fifo_threshold() | Set an RX FIFO threshold. |
| | hal_qspi_get_tx_fifo_threshold() | Get a TX FIFO threshold. |
| | hal_qspi_get_rx_fifo_threshold() | Get an RX FIFO threshold. |
| Sleep | hal_qspi_suspend_reg() | Suspend registers related to QSPI configuration in sleep mode. |
| | hal_qspi_resume_reg() | Resume registers related to QSPI configuration during wakeup. |

The sections below elaborate on these APIs.

2.17.4.1 hal_qspi_init

Table 2-238 hal_qspi_init API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_qspi_init(qspi_handle_t *p_qspi) |
| Function Description | Initialize QSPI mode and relevant handles based on parameters specified in qspi_init_t . |
| Parameter | p_qspi: pointer to variables of qspi_handle_t . The variable contains the configuration information of a specified QSPI. |
| Return Value | HAL status |
| Remarks | |

2.17.4.2 hal_qspi_deinit

Table 2-239 hal_qspi_deinit API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_qspi_deinit(qspi_handle_t *p_qspi) |
| Function Description | Deinitialize the QSPI peripheral. |
| Parameter | p_qspi: pointer to variables of qspi_handle_t . The variable contains the configuration information of a specified QSPI. |
| Return Value | HAL status |
| Remarks | |

2.17.4.3 hal_qspi_msp_init

Table 2-240 hal_qspi_msp_init API

| | |
|-----------------------------|---|
| Function Prototype | void hal_qspi_msp_init(qspi_handle_t *p_qspi) |
| Function Description | Initialize GPIO pin multiplexing, NVIC interrupts, and DMA channels used by QSPI. |
| Parameter | p_qspi: pointer to variables of qspi_handle_t . The variable contains the configuration information of a specified QSPI. |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite the API to initialize GPIO pin multiplexing, NVIC interrupts, and DMA channels. |

2.17.4.4 hal_qspi_msp_deinit

Table 2-241 hal_qspi_msp_deinit API

| | |
|-----------------------------|---|
| Function Prototype | void hal_qspi_msp_deinit(qspi_handle_t *p_qspi) |
| Function Description | Deinitialize GPIO pin multiplexing, NVIC interrupts, and DMA channels used by QSPI. |
| Parameter | p_qspi: pointer to variables of qspi_handle_t . The variable contains the configuration information of a specified QSPI. |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite the API to deinitialize GPIO pin multiplexing, NVIC interrupts, and DMA channels. |

2.17.4.5 hal_qspi_command_transmit

Table 2-242 hal_qspi_command_transmit API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_qspi_command_transmit(qspi_handle_t *p_qspi, qspi_command_t *p_cmd, uint8_t *p_data, uint32_t timeout) |
| Function Description | Transmit data (including instructions, addresses, and data) in polling mode. |

| | |
|---------------------|---|
| Parameter | <p>p_qspi: pointer to variables of qspi_handle_t. The variable contains the configuration information of a specified QSPI.</p> <p>p_cmd: pointer to variables of qspi_command_t. The variable contains the configuration information for commands.</p> <p>p_data: pointer to data buffer</p> <p>timeout: timeout period</p> |
| Return Value | HAL status |
| Remarks | |

2.17.4.6 hal_qspi_command_receive

Table 2-243 hal_qspi_command_receive API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_qspi_command_receive(qspi_handle_t *p_qspi, qspi_command_t *p_cmd, uint8_t *p_data, uint32_t timeout) |
| Function Description | Receive data (including instructions, addresses, dummy, and data) in polling mode. |
| Parameter | <p>p_qspi: pointer to variables of qspi_handle_t. The variable contains the configuration information of a specified QSPI.</p> <p>p_cmd: pointer to variables of qspi_command_t. The variable contains the configuration information for commands.</p> <p>p_data: pointer to data buffer</p> <p>timeout: timeout period</p> |
| Return Value | HAL status |
| Remarks | |

2.17.4.7 hal_qspi_command

Table 2-244 hal_qspi_command API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_qspi_command(qspi_handle_t *p_qspi, qspi_command_t *p_cmd, uint32_t timeout) |
| Function Description | Transmit instructions in polling mode. |
| Parameter | <p>p_qspi: pointer to variables of qspi_handle_t. The variable contains the configuration information of a specified QSPI.</p> <p>p_cmd: pointer to variables of qspi_command_t. The variable contains the configuration information for commands.</p> <p>timeout: timeout period</p> |
| Return Value | HAL status |

| | |
|----------------|---|
| Remarks | This function is used to transmit instructions in polling mode only. The function can be used in association with hal_qspi_transmit() and hal_qspi_receive() , to transmit and receive instructions, addresses, and data. You can also use hal_qspi_command_transmit() and hal_qspi_command_receive() directly for such transmission and reception. |
|----------------|---|

2.17.4.8 hal_qspi_transmit

Table 2-245 hal_qspi_transmit API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_qspi_transmit(qspi_handle_t *p_qspi, uint8_t *p_data, uint32_t length, uint32_t timeout) |
| Function Description | Transmit a large volume of data (in SPI mode) in polling mode. |
| Parameter | p_qspi: pointer to variables of qspi_handle_t . The variable contains the configuration information of a specified QSPI. p_data: pointer to data buffer length: length of data to be transmitted timeout: timeout period |
| Return Value | HAL status |
| Remarks | This function can only be executed in Standard SPI mode. |

2.17.4.9 hal_qspi_receive

Table 2-246 hal_qspi_receive API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_qspi_receive(qspi_handle_t *p_qspi, uint8_t *p_data, uint32_t length, uint32_t timeout) |
| Function Description | Receive a large amount of data (in SPI mode) in polling mode. |
| Parameter | p_qspi: pointer to variables of qspi_handle_t . The variable contains the configuration information of a specified QSPI. p_data: pointer to data buffer length: length of data to be received timeout: timeout period |
| Return Value | HAL status |
| Remarks | This function can only be executed in Standard SPI mode. |

2.17.4.10 hal_qspi_command_transmit_it

Table 2-247 hal_qspi_command_transmit_it API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_qspi_command_transmit_it(qspi_handle_t *p_qspi, qspi_command_t *p_cmd, uint8_t *p_data) |
| Function Description | Transmit data (including instructions, addresses, and data) in interrupt mode. |
| Parameter | p_qspi: pointer to variables of qspi_handle_t . The variable contains the configuration information of a specified QSPI. p_cmd: pointer to variables of qspi_command_t . The variable contains the configuration information for commands. p_data: pointer to data buffer |
| Return Value | HAL status |
| Remarks | During transmission, if the QSPI interrupt handler cannot respond in time, a data transmission error may occur. |

2.17.4.11 hal_qspi_command_receive_it

Table 2-248 hal_qspi_command_receive_it API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_qspi_command_receive_it(qspi_handle_t *p_qspi, qspi_command_t *p_cmd, uint8_t *p_data) |
| Function Description | Receive data (including instructions, addresses, dummy, and data) in interrupt mode. |
| Parameter | p_qspi: pointer to variables of qspi_handle_t . The variable contains the configuration information of a specified QSPI. p_cmd: pointer to variables of qspi_command_t . The variable contains the configuration information for commands. p_data: pointer to data buffer |
| Return Value | HAL status |
| Remarks | During transmission, if the QSPI interrupt handler cannot respond in time, a data reception error may occur. |

2.17.4.12 hal_qspi_command_it

Table 2-249 hal_qspi_command_it API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_qspi_command_it(qspi_handle_t *p_qspi, qspi_command_t *p_cmd) |
| Function Description | Transmit instructions in interrupt mode. |
| Parameter | p_qspi: pointer to variables of qspi_handle_t . The variable contains the configuration information of a specified QSPI. |

| | |
|---------------------|---|
| | p_cmd: pointer to variables of qspi_command_t . The variable contains the configuration information for commands. |
| Return Value | HAL status |
| Remarks | This function is used to transmit instructions in interrupt mode only. The function can be used in association with hal_qspi_receive_it() and hal_qspi_transmit_it() , to transmit and receive instructions, addresses, and data. You can also use hal_qspi_command_receive_it() and hal_qspi_command_transmit_it() directly for such transmission and reception. |

2.17.4.13 hal_qspi_transmit_it

Table 2-250 hal_qspi_transmit_it API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_qspi_transmit_it(qspi_handle_t *p_qspi, uint8_t *p_data, uint32_t length) |
| Function Description | Transmit data (in SPI mode) in interrupt mode. |
| Parameter | p_qspi: pointer to variables of qspi_handle_t . The variable contains the configuration information of a specified QSPI. p_data: pointer to data buffer length: length of data to be transmitted |
| Return Value | HAL status |
| Remarks | <ul style="list-style-type: none"> This function can only be executed in Standard SPI mode. During transmission, if the QSPI interrupt handler cannot respond in time, a data transmission error may occur. |

2.17.4.14 hal_qspi_receive_it

Table 2-251 hal_qspi_receive_it API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_qspi_receive_it(qspi_handle_t *p_qspi, uint8_t *p_data, uint32_t length) |
| Function Description | Receive data (in SPI mode) in interrupt mode. |
| Parameter | p_qspi: pointer to variables of qspi_handle_t . The variable contains the configuration information of a specified QSPI. p_data: pointer to data buffer length: length of data to be received |
| Return Value | HAL status |
| Remarks | <ul style="list-style-type: none"> This function can only be executed in Standard SPI mode. During transmission, if the QSPI interrupt handler cannot respond in time, a data reception error may occur. |

2.17.4.15 hal_qspi_command_transmit_dma

Table 2-252 hal_qspi_command_transmit_dma API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_qspi_command_transmit_dma(qspi_handle_t *p_qspi, qspi_command_t *p_cmd, uint8_t *p_data) |
| Function Description | Transmit data (including instructions, addresses, and data) in DMA mode. |
| Parameter | p_qspi: pointer to variables of qspi_handle_t . The variable contains the configuration information of a specified QSPI. p_cmd: pointer to variables of qspi_command_t . The variable contains the configuration information for commands. p_data: pointer to data buffer |
| Return Value | HAL status |
| Remarks | |

2.17.4.16 hal_qspi_command_receive_dma

Table 2-253 hal_qspi_command_receive_dma API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_qspi_command_receive_dma(qspi_handle_t *p_qspi, qspi_command_t *p_cmd, uint8_t *p_data) |
| Function Description | Receive data (including instructions, addresses, dummy, and data) in DMA mode. |
| Parameter | p_qspi: pointer to variables of qspi_handle_t . The variable contains the configuration information of a specified QSPI. p_cmd: pointer to variables of qspi_command_t . The variable contains the configuration information for commands. p_data: pointer to data buffer |
| Return Value | HAL status |
| Remarks | |

2.17.4.17 hal_qspi_command_dma

Table 2-254 hal_qspi_command_dma API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_qspi_command_dma(qspi_handle_t *p_qspi, qspi_command_t *p_cmd) |
| Function Description | Transmit instructions in DMA mode. |
| Parameter | p_qspi: pointer to variables of qspi_handle_t . The variable contains the configuration information of a specified QSPI. p_cmd: pointer to variables of qspi_command_t . The variable contains the configuration information for commands. |
| Return Value | HAL status |

| | |
|----------------|---|
| Remarks | This function is used to transmit instructions in DMA mode only. The function can be used in association with hal_qspi_receive_dma() and hal_qspi_transmit_dma() , to transmit and receive instructions, addresses, and data. You can also use hal_qspi_command_receive_dma() and hal_qspi_command_transmit_dma() directly for such transmission and reception. |
|----------------|---|

2.17.4.18 hal_qspi_transmit_dma

Table 2-255 hal_qspi_transmit_dma API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_qspi_transmit_dma(qspi_handle_t *p_qspi, uint8_t *p_data, uint32_t length) |
| Function Description | Transmit data (in SPI mode) in DMA mode. |
| Parameter | <p>p_qspi: pointer to variables of qspi_handle_t. The variable contains the configuration information of a specified QSPI.</p> <p>p_data: pointer to data buffer</p> <p>length: length of data to be transmitted</p> |
| Return Value | HAL status |
| Remarks | <p>This function can only be executed in Standard SPI mode:</p> <ul style="list-style-type: none"> • If access in nippie mode from the DMA peripheral is configured, the data size and the FIFO threshold shall be half-byte aligned. • If access in byte mode from the DMA peripheral is configured, the data size and the FIFO threshold shall be byte aligned. |

2.17.4.19 hal_qspi_receive_dma

Table 2-256 hal_qspi_receive_dma API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_qspi_receive_dma(qspi_handle_t *p_qspi, uint8_t *p_data, uint32_t length) |
| Function Description | Receive data (in SPI mode) in DMA mode. |
| Parameter | <p>p_qspi: pointer to variables of qspi_handle_t. The variable contains the configuration information of a specified QSPI.</p> <p>p_data: pointer to data buffer</p> <p>length: length of data to be received</p> |
| Return Value | HAL status |
| Remarks | <p>This function can only be executed in Standard SPI mode:</p> <ul style="list-style-type: none"> • If access in nippie mode from the DMA peripheral is configured, the data size and the FIFO threshold shall be half-byte aligned. • If access in byte mode from the DMA peripheral is configured, the data size and the FIFO threshold shall be byte aligned. |

2.17.4.20 hal_qspi_abort

Table 2-257 hal_qspi_abort API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_qspi_abort(qspi_handle_t *p_qspi) |
| Function Description | In polling mode, abort data transfer in interrupt/DMA mode. |
| Parameter | p_qspi: pointer to variables of qspi_handle_t . The variable contains the configuration information of a specified QSPI. |
| Return Value | HAL status |
| Remarks | |

2.17.4.21 hal_qspi_abort_it

Table 2-258 hal_qspi_abort_it API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_qspi_abort_it(qspi_handle_t *p_qspi) |
| Function Description | In interrupt mode, abort data transfer in interrupt/DMA mode. |
| Parameter | p_qspi: pointer to variables of qspi_handle_t . The variable contains the configuration information of a specified QSPI. |
| Return Value | HAL status |
| Remarks | |

2.17.4.22 hal_qspi_irq_handler

Table 2-259 hal_qspi_irq_handler API

| | |
|-----------------------------|--|
| Function Prototype | void hal_qspi_irq_handler(qspi_handle_t *p_qspi) |
| Function Description | Handle QSPI interrupt requests. |
| Parameter | p_qspi: pointer to variables of qspi_handle_t . The variable contains the configuration information of a specified QSPI. |
| Return Value | None |
| Remarks | |

2.17.4.23 hal_qspi_tx_cplt_callback

Table 2-260 hal_qspi_tx_cplt_callback API

| | |
|-----------------------------|--|
| Function Prototype | void hal_qspi_tx_cplt_callback(qspi_handle_t *p_qspi) |
| Function Description | Transmission complete callback function |
| Parameter | p_qspi: pointer to variables of qspi_handle_t . The variable contains the configuration information of a specified QSPI. |

| | |
|---------------------|--|
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite this function before using it. |

2.17.4.24 hal_qspi_rx_cplt_callback

Table 2-261 hal_qspi_rx_cplt_callback API

| | |
|-----------------------------|--|
| Function Prototype | void hal_qspi_rx_cplt_callback(qspi_handle_t *p_qspi) |
| Function Description | Reception complete callback function |
| Parameter | p_qspi: pointer to variables of qspi_handle_t . The variable contains the configuration information of a specified QSPI. |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite this function before using it. |

2.17.4.25 hal_qspi_error_callback

Table 2-262 hal_qspi_error_callback API

| | |
|-----------------------------|--|
| Function Prototype | void hal_qspi_error_callback(qspi_handle_t *p_qspi) |
| Function Description | Transfer error callback function |
| Parameter | p_qspi: pointer to variables of qspi_handle_t . The variable contains the configuration information of a specified QSPI. |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite this function before using it. |

2.17.4.26 hal_qspi_abort_cplt_callback

Table 2-263 hal_qspi_abort_cplt_callback API

| | |
|-----------------------------|--|
| Function Prototype | void hal_qspi_abort_cplt_callback(qspi_handle_t *p_qspi) |
| Function Description | Abort complete callback function |
| Parameter | p_qspi: pointer to variables of qspi_handle_t . The variable contains the configuration information of a specified QSPI. |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite this function before using it. |

2.17.4.27 hal_qspi_get_state

Table 2-264 hal_qspi_get_state API

| | |
|-----------------------------|---|
| Function Prototype | hal_qspi_state_t hal_qspi_get_state(qspi_handle_t *p_qspi) |
| Function Description | Get the QSPI operating state. |
| Parameter | p_qspi: pointer to variables of qspi_handle_t . The variable contains the configuration information of a specified QSPI. |
| Return Value | QSPI operating state. This parameter can be one of the following values: <ul style="list-style-type: none"> • HAL_QSPI_STATE_RESET (not initialized) • HAL_QSPI_STATE_READY (initialized and ready for use) • HAL_QSPI_STATE_BUSY (busy) • HAL_QSPI_STATE_BUSY_INDIRECT_TX (TX ongoing) • HAL_QSPI_STATE_BUSY_INDIRECT_RX (RX ongoing) • HAL_QSPI_STATE_ABORT (aborted) • HAL_QSPI_STATE_ERROR (error) |
| Remarks | |

2.17.4.28 hal_qspi_get_error

Table 2-265 hal_qspi_get_error API

| | |
|-----------------------------|---|
| Function Prototype | uint32_t hal_qspi_get_error(qspi_handle_t *p_qspi) |
| Function Description | Return the QSPI error code. |
| Parameter | p_qspi: pointer to variables of qspi_handle_t . The variable contains the configuration information of a specified QSPI. |
| Return Value | QSPI error code. This parameter can be one of the following values: <ul style="list-style-type: none"> • HAL_QSPI_ERROR_NONE (no error) • HAL_QSPI_ERROR_TIMEOUT (timeout) • HAL_QSPI_ERROR_TRANSFER (transfer error) • HAL_QSPI_ERROR_DMA (DMA transfer error) • HAL_QSPI_ERROR_INVALID_PARAM (invalid parameter) |
| Remarks | |

2.17.4.29 hal_qspi_set_timeout

Table 2-266 hal_qspi_set_timeout API

| | |
|---------------------------|--|
| Function Prototype | void hal_qspi_set_timeout(qspi_handle_t *p_qspi, uint32_t timeout) |
|---------------------------|--|

| | |
|-----------------------------|--|
| Function Description | Set a timeout period for QSPI operations. |
| Parameter | p_qspi: pointer to variables of qspi_handle_t . The variable contains the configuration information of a specified QSPI. timeout: timeout period for QSPI memory access |
| Return Value | None |
| Remarks | |

2.17.4.30 hal_qspi_set_tx_fifo_threshold

Table 2-267 hal_qspi_set_tx_fifo_threshold API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_qspi_set_tx_fifo_threshold(qspi_handle_t *p_qspi, uint32_t threshold) |
| Function Description | Set a QSPI TX FIFO threshold. |
| Parameter | p_qspi: pointer to variables of qspi_handle_t . The variable contains the configuration information of a specified QSPI. threshold: TX FIFO threshold (value range: 0 to 7; 0 indicates that a TX FIFO is empty, and 7 indicates that a TX FIFO reaches 1 byte minus the full threshold.) |
| Return Value | HAL status |
| Remarks | |

2.17.4.31 hal_qspi_set_rx_fifo_threshold

Table 2-268 hal_qspi_set_rx_fifo_threshold API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_qspi_set_rx_fifo_threshold(qspi_handle_t *p_qspi, uint32_t threshold) |
| Function Description | Set a QSPI RX FIFO threshold. |
| Parameter | p_qspi: pointer to variables of qspi_handle_t . The variable contains the configuration information of a specified QSPI. threshold: RX FIFO threshold (value range: 0 to 7; 0 indicates that an RX FIFO is 1 byte, and 7 indicates that an RX FIFO is full.) |
| Return Value | HAL status |
| Remarks | |

2.17.4.32 hal_qspi_get_tx_fifo_threshold

Table 2-269 hal_qspi_get_tx_fifo_threshold API

| | |
|-----------------------------|--|
| Function Prototype | uint32_t hal_qspi_get_tx_fifo_threshold(qspi_handle_t *p_qspi) |
| Function Description | Get a QSPI TX FIFO threshold. |

| | |
|---------------------|---|
| Parameter | p_qspi: pointer to variables of qspi_handle_t . The variable contains the configuration information of a specified QSPI. |
| Return Value | TX FIFO threshold (value range: 0 to 7; 0 indicates that a TX FIFO is empty, and 7 indicates that a TX FIFO reaches 1 byte minus the full threshold.) |
| Remarks | |

2.17.4.33 hal_qspi_get_rx_fifo_threshold

Table 2-270 hal_qspi_get_rx_fifo_threshold API

| | |
|-----------------------------|--|
| Function Prototype | uint32_t hal_qspi_get_rx_fifo_threshold(qspi_handle_t *p_qspi) |
| Function Description | Get a QSPI RX FIFO threshold. |
| Parameter | p_qspi: pointer to variables of qspi_handle_t . The variable contains the configuration information of a specified QSPI. |
| Return Value | RX FIFO threshold (value range: 0 to 7; 0 indicates that an RX FIFO is 1 byte, and 7 indicates that an RX FIFO is full.) |
| Remarks | |

2.17.4.34 hal_qspi_suspend_reg

Table 2-271 hal_qspi_suspend_reg API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_qspi_suspend_reg(qspi_handle_t *p_qspi) |
| Function Description | Suspend registers related to QSPI configuration in sleep mode. |
| Parameter | p_qspi: pointer to variables of qspi_handle_t . The variable contains the configuration information of a specified QSPI. |
| Return Value | HAL status |
| Remarks | |

2.17.4.35 hal_qspi_resume_reg

Table 2-272 hal_qspi_resume_reg API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_qspi_resume_reg(qspi_handle_t *p_qspi) |
| Function Description | Resume registers related to QSPI configuration during wakeup. |
| Parameter | p_qspi: pointer to variables of qspi_handle_t . The variable contains the configuration information of a specified QSPI. |
| Return Value | HAL status |
| Remarks | |

2.18 HAL PWM Generic Driver

2.18.1 PWM Driver Functionalities

The HAL Pulse Width Modulation (PWM) driver features the following functionalities:

- The clock frequency can be as high as that of the system clock.
- Two PWM modules, each with 3 output channels
- Configurable output frequency which can be dynamically updated
- Two output modes: fixed duty ratio mode and breathing mode (cyclic change of duty ratio: 0 → 100% → 0)
- Two alignment modes: the duty ratio in each cycle can be left-edge-aligned or center-aligned.
- Output pause

2.18.2 How to Use PWM Driver

Developers can use the PWM driver in the following scenarios:

1. Declare a `pwm_handle_t` handle structure variable, for example: `pwm_handle_t pwm_handle`.
2. Initialize the PWM low-level resources by overwriting `hal_pwm_msp_init()`: Configure the mode for GPIO pins of PWM channels as `GPIO_PIN_MUX` (multiplexing mode) by calling `hal_gpio_int()`, and set the multiplexing mode.
3. Configure the output mode, the alignment mode, the output frequency, and the output channel for the init structure of `pwm_handle`. For common duty ratio mode, the channel duty ratio and the output polarity shall also be configured; for breathing mode, the breath and hold periods shall also be configured.
4. Configure PWM registers by calling `hal_pwm_init(&pwm_handle)`. During configuration, `hal_pwm_init()` automatically calls the overwritten `hal_pwm_msp_init()`, to initialize GPIO pins and other low-level resources for PWM.
5. Declare a `pwm_channel_init_t` channel initialization structure variable, for example: `pwm_channel_init_t channel_init`.
6. Configure the channel duty ratio and the output polarity for `channel_init` based on the output mode:
 - Fixed duty ratio mode: Configure the channel duty ratio and the output polarity.
 - Breathing mode: Only configure the channel output polarity.
7. Call `hal_pwm_config_channel(&pwm_handle,&channel_init,HAL_PWM_ACTIVE_CHANNEL_x)` to configure the output channel `HAL_PWM_ACTIVE_CHANNEL_x`, where `x` can be A, B, C, or ALL.
8. Start PWM output by calling `hal_pwm_start()`.
9. Stop PWM output by calling `hal_pwm_stop()`. Developers can also modify the output channel configurations by calling `hal_pwm_config_channel()`.

2.18.3 PWM Driver Structures

2.18.3.1 pwm_channel_init_t

The channel description structure `pwm_channel_init_t` of the PWM driver is defined below:

Table 2-273 `pwm_channel_init_t` structure

| Data Field | Field Description | Value |
|-------------------------------------|-------------------|---|
| <code>uint8_t duty</code> | Duty ratio | 0 to 100 |
| <code>uint8_t drive_polarity</code> | Driver polarity | This parameter can be one of the following values: <ul style="list-style-type: none"> PWM_DRIVEPOLARITY_NEGATIVE (negative driver polarity) PWM_DRIVEPOLARITY_POSITIVE (positive driver polarity) |

2.18.3.2 pwm_init_t

The initialization structure `pwm_init_t` of the PWM driver is defined below:

Table 2-274 `pwm_init_t` structure

| Data Field | Field Description | Value |
|-------------------------------|-----------------------|---|
| <code>uint32_t mode</code> | PWM output mode | This parameter can be one of the following values: <ul style="list-style-type: none"> PWM_MODE_FLICKER (fixed duty ratio mode) PWM_MODE_BREATH (breathing mode) |
| <code>uint32_t align</code> | PWM alignment mode | This parameter can be one of the following values: <ul style="list-style-type: none"> PWM_ALIGNED_EDGE (left-edge-aligned) PWM_ALIGNED_CENTER (center-aligned) |
| <code>uint32_t freq</code> | PWM output frequency | 1 to 32000000; recommended frequency: lower than 500000 |
| <code>uint32_t bperiod</code> | Breathing period | 1 to 67108 (64 MHz) 1 to 89478 (48 MHz) 1 to 134217 (32 MHz) 1 to 178956 (24 MHz) 1 to 268435 (16 MHz) Unit: ms |
| <code>uint32_t hperiod</code> | Breath holding period | 1 to 262 (64 MHz) 1 to 349 (48 MHz) 1 to 524 (32 MHz) 1 to 699 (24 MHz) 1 to 1048 (16 MHz) Unit: ms |

| Data Field | Field Description | Value |
|------------------------------|--|--|
| pwm_channel_init_t channel_a | Configuration parameter for output channel A | See " Section 2.18.3.1 pwm_channel_init_t ". |
| pwm_channel_init_t channel_b | Configuration parameter for output channel B | See " Section 2.18.3.1 pwm_channel_init_t ". |
| pwm_channel_init_t channel_c | Configuration parameter for output channel C | See " Section 2.18.3.1 pwm_channel_init_t ". |

2.18.3.3 pwm_handle_t

The handle structure pwm_handle_t of the PWM driver is defined below:

Table 2-275 pwm_handle_t structure

| Data Field | Field Description | Value |
|---|--|---|
| pwm_regs_t *p_instance | PWM peripheral instance | This parameter can be one of the following values: <ul style="list-style-type: none"> • PWM0 • PWM1 |
| pwm_init_t init | PWM initialization structure | See " Section 2.18.3.2 pwm_init_t ". |
| hal_pwm_active_channel_t active_channel | Enable PWM output channels. | This parameter can be one of the following values: <ul style="list-style-type: none"> • HAL_PWM_ACTIVE_CHANNEL_A • HAL_PWM_ACTIVE_CHANNEL_B • HAL_PWM_ACTIVE_CHANNEL_C • HAL_PWM_ACTIVE_CHANNEL_ALL • HAL_PWM_ACTIVE_CHANNEL_CLEARED |
| __IO hal_lock_t lock | PWM lock (managed by PWM driver and initialization by developers not required) | N/A |
| __IO hal_pwm_state_t state | PWM operating state (initialization by developers not required) | This parameter can be one of the following values: <ul style="list-style-type: none"> • HAL_PWM_STATE_RESET (not initialized) • HAL_PWM_STATE_READY (initialized and ready for use) • HAL_PWM_STATE_BUSY (busy) • HAL_PWM_STATE_ERROR (error) |
| uint32_t retention[11] | PWM register information (managed by PWM driver and initialization by developers not required) | N/A |

2.18.4 PWM Driver APIs

The PWM driver APIs are listed in the table below:

Table 2-276 PWM driver APIs

| API Type | API Name | Description |
|-----------------|--------------------------|---|
| Initialization | hal_pwm_init() | Initialize the PWM peripheral, and configure output parameters. |
| | hal_pwm_deinit() | Deinitialize the PWM peripheral. |
| | hal_pwm_msp_init() | Initialize GPIOs used by the PWM peripheral. |
| | hal_pwm_msp_deinit() | Deinitialize GPIOs used by the PWM peripheral. |
| I/O operation | hal_pwm_start() | Start PWM output. |
| | hal_pwm_stop() | Stop PWM output. |
| | hal_pwm_update_freq() | Update PWM output frequency. |
| | hal_pwm_config_channel() | Configure PWM channel parameters. |
| State and error | hal_pwm_get_state() | Get the driver operating state. |
| Sleep | hal_pwm_suspend_reg() | Suspend registers related to PWM configuration in sleep mode. |
| | hal_pwm_resume_reg() | Resume registers related to PWM configuration during wakeup. |

The sections below elaborate on these APIs.

2.18.4.1 hal_pwm_init

Table 2-277 hal_pwm_init API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_pwm_init(pwm_handle_t *p_pwm) |
| Function Description | Initialize the PWM peripheral and related handles according to parameters of pwm_init_t . |
| Parameter | p_pwm: pointer to variables of pwm_handle_t . The variable contains the configuration information of a specified PWM. |
| Return Value | HAL status |
| Remarks | |

2.18.4.2 hal_pwm_deinit

Table 2-278 hal_pwm_deinit API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_pwm_deinit(pwm_handle_t *p_pwm) |
| Function Description | Deinitialize the PWM peripheral. |

| | |
|---------------------|---|
| Parameter | p_pwm: pointer to variables of pwm_handle_t . The variable contains the configuration information of a specified PWM. |
| Return Value | HAL status |
| Remarks | |

2.18.4.3 hal_pwm_msp_init

Table 2-279 hal_pwm_msp_init API

| | |
|-----------------------------|--|
| Function Prototype | void hal_pwm_msp_init(pwm_handle_t *p_pwm) |
| Function Description | Initialize GPIOs used by the PWM peripheral. |
| Parameter | p_pwm: pointer to variables of pwm_handle_t |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite the API to initialize GPIOs. |

2.18.4.4 hal_pwm_msp_deinit

Table 2-280 hal_pwm_msp_deinit API

| | |
|-----------------------------|--|
| Function Prototype | void hal_pwm_msp_deinit(pwm_handle_t *p_pwm) |
| Function Description | Deinitialize GPIOs used by the PWM peripheral. |
| Parameter | p_pwm: pointer to variables of pwm_handle_t |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite the API to deinitialize GPIOs. |

2.18.4.5 hal_pwm_start

Table 2-281 hal_pwm_start API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_pwm_start(pwm_handle_t *p_pwm) |
| Function Description | Start PWM output. |
| Parameter | p_pwm: pointer to variables of pwm_handle_t . The variable contains the configuration information of a specified PWM. |
| Return Value | HAL status |
| Remarks | Call this function to output PWM channel waveform after the output channel parameters are configured. |

2.18.4.6 hal_pwm_stop

Table 2-282 hal_pwm_stop API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_pwm_stop(pwm_handle_t *p_pwm) |
| Function Description | Stop PWM output. |
| Parameter | p_pwm: pointer to variables of pwm_handle_t . The variable contains the configuration information of a specified PWM. |
| Return Value | HAL status |
| Remarks | During PWM waveform output, call this function to stop output. |

2.18.4.7 hal_pwm_update_freq

Table 2-283 hal_pwm_update_freq API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_pwm_update_freq(pwm_handle_t *p_pwm, uint32_t freq) |
| Function Description | Update PWM output frequency. |
| Parameter | p_pwm: pointer to variables of pwm_handle_t . The variable contains the configuration information of a specified PWM. freq: output frequency Range: 1 to 32000000; recommended frequency: lower than 500000 |
| Return Value | HAL status |
| Remarks | During PWM waveform output, call this function to change the PWM waveform output frequency. |

2.18.4.8 hal_pwm_config_channel

Table 2-284 hal_pwm_config_channel API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_pwm_config_channel(pwm_handle_t *p_pwm, pwm_channel_init_t *p_config, hal_pwm_active_channel_t channel) |
| Function Description | Configure PWM output channels. |
| Parameter | p_pwm: pointer to variables of pwm_handle_t . The variable contains the configuration information of a specified PWM. p_config: pointer to variables of pwm_channel_init_t . The variable contains the configuration information of PWM channels. channel: channels to be configured |
| Return Value | HAL status |
| Remarks | When PWM waveform output stops, call this function to reconfigure the channel parameters. |

2.18.4.9 hal_pwm_get_state

Table 2-285 hal_pwm_get_state API

| | |
|---------------------------|--|
| Function Prototype | hal_pwm_state_t hal_pwm_get_state(pwm_handle_t *p_pwm) |
|---------------------------|--|

| | |
|-----------------------------|--|
| Function Description | Get the PWM operating state. |
| Parameter | p_pwm: pointer to variables of pwm_handle_t . The variable contains the configuration information of a specified PWM. |
| Return Value | PWM operating state. This parameter can be one of the following values: <ul style="list-style-type: none"> • HAL_PWM_STATE_RESET (not initialized) • HAL_PWM_STATE_READY (initialized and ready for use) • HAL_PWM_STATE_BUSY (busy) • HAL_PWM_STATE_ERROR (error) |
| Remarks | |

2.18.4.10 hal_pwm_suspend_reg

Table 2-286 hal_pwm_suspend_reg API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_pwm_suspend_reg(pwm_handle_t *p_pwm) |
| Function Description | Suspend registers related to PWM configuration in sleep mode. |
| Parameter | p_pwm: pointer to variables of pwm_handle_t . The variable contains the configuration information of a specified PWM. |
| Return Value | PWM operating state. This parameter can be one of the following values: <ul style="list-style-type: none"> • HAL_PWM_STATE_RESET (not initialized) • HAL_PWM_STATE_READY (initialized and ready for use) • HAL_PWM_STATE_BUSY (busy) • HAL_PWM_STATE_ERROR (error) |
| Remarks | |

2.18.4.11 hal_pwm_resume_reg

Table 2-287 hal_pwm_resume_reg API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_pwm_resume_reg(pwm_handle_t *p_pwm) |
| Function Description | Resume registers related to PWM configuration during wakeup. |
| Parameter | p_pwm: pointer to variables of pwm_handle_t . The variable contains the configuration information of a specified PWM. |
| Return Value | PWM operating state. This parameter can be one of the following values: <ul style="list-style-type: none"> • HAL_PWM_STATE_RESET (not initialized) • HAL_PWM_STATE_READY (initialized and ready for use) • HAL_PWM_STATE_BUSY (busy) • HAL_PWM_STATE_ERROR (error) |

| | |
|---------|--|
| Remarks | |
|---------|--|

2.19 HAL PWR Generic Driver

2.19.1 PWR Driver Functionalities

The HAL Power Controller (PWR) driver features the following functionalities:

- Six modes for wakeup from ultra deep sleep status: AON_GPIO, AON SLEEP TIMER, BLE_TIMER, CALENDAR, COMP, and BOD
- Four AON GPIO wakeup approaches: high level, low level, rising edge, and falling edge
- Configurable wakeup time for AON SLEEP TIMER
- Power management and mode switching for Bluetooth LE Core and Bluetooth LE Timer

2.19.2 How to Use PWR Driver

The PWM driver controls the power mode and the ultra deep sleep mode of Bluetooth LE Core, and Bluetooth LE Timer in MCU. Developers can execute related APIs on demand.

2.19.2.1 Bluetooth LE Power Configuration

GR551x SoCs support power management for Bluetooth LE Core and Bluetooth LE Timer, and the supported power status includes: Power On and Power Down.

- Power On: Bluetooth LE Core or Bluetooth LE Timer is powered on and runs normally.
- Power Down: Bluetooth LE Core or Bluetooth LE Timer is powered down and stops running.

In addition, GR551x SoCs support reset mode and operating mode for Bluetooth LE Core and Bluetooth LE Timer. Bluetooth LE Core and Bluetooth LE Timer can switch between the two modes.

You can call `hal_pwr_set_comm_power()` to manage power for Bluetooth LE Core and Bluetooth LE Timer, and call `hal_pwr_set_comm_mode()` for mode switching.

2.19.2.2 Ultra Deep Sleep Configuration

GR551x SoCs support ultra deep sleep mode. In this mode, all peripherals and the Bluetooth LE Core in the MCU subsystem are powered down, and the SoC is in low-power mode.

Before the system enters the ultra deep sleep mode, wakeup conditions are required, including: External, Timer, Bluetooth LE, and External + Timer + Bluetooth LE:

- External: The system can be awoken by AON GPIOs; pins and types for wakeup are required.
- Timer: The system can be awoken by AON SLEEP TIMER; intervals to wake up MCU are required, and the clock frequency for the AON SLEEP TIMER is 40 kHz.
- Bluetooth LE : The system can be awoken by Bluetooth LE TIMER.

- External + Timer + Bluetooth LE: The system can be awoken by AON GPIOs, AON SLEEP TIMER, or Bluetooth LE TIMER; pins, types, and intervals to wake up MCU are required.

You can call `hal_pwr_set_wakeup_condition()` to configure wakeup conditions; if External is included in the wakeup conditions, call `hal_pwr_config_timer_wakeup()` to configure AON GPIO pins and wakeup types; if Timer is included in the wakeup conditions, call `hal_pwr_config_ext_wakeup()` to configure the count for AON SLEEP TIMER.

2.19.3 PWR Driver APIs

The PWR driver APIs are listed in the table below:

Table 2-288 PWR driver APIs

| API Type | API Name | Description |
|---------------------------------|---|--|
| Control | <code>hal_pwr_set_wakeup_condition()</code> | Set wakeup conditions for ultra deep sleep mode. |
| | <code>hal_pwr_config_timer_wakeup()</code> | Configure wakeup parameters for AON Sleep Timer. |
| | <code>hal_pwr_config_ext_wakeup()</code> | Configure wakeup parameters for AON GPIO. |
| | <code>hal_pwr_set_comm_power()</code> | Set power status for Bluetooth LE Core and Bluetooth LE Timer. |
| | <code>hal_pwr_set_comm_mode()</code> | Set the mode for Bluetooth LE Core and Bluetooth LE Timer. |
| | <code>hal_pwr_enter_chip_deepsleep()</code> | Enter ultra deep sleep mode. |
| | <code>hal_pwr_get_timer_current_value()</code> | Get the current timer value. |
| | <code>hal_pwr_disable_ext_wakeup()</code> | Disable the specified AON GPIO wakeup system. |
| Interrupt handling and callback | <code>hal_pwr_sleep_timer_irq_handler()</code> | SleepTimer interrupt handler |
| | <code>hal_pwr_sleep_timer_elapsed_callback()</code> | SleepTimer interrupt callback function |

The sections below elaborate on these APIs.

2.19.3.1 `hal_pwr_set_wakeup_condition`

Table 2-289 `hal_pwr_set_wakeup_condition` API

| | |
|-----------------------------|---|
| Function Prototype | <code>void hal_pwr_set_wakeup_condition(uint32_t condition)</code> |
| Function Description | Set wakeup conditions for ultra deep sleep mode. |
| Parameter | <p>condition: wakeup conditions. This parameter can be one of the following values:</p> <ul style="list-style-type: none"> PWR_WKUP_COND_EXT (AON_GPIO) PWR_WKUP_COND_TIMER (AON Sleep Timer) PWR_WKUP_COND_BLE (Bluetooth LE Timer) |

| | |
|---------------------|--|
| | <ul style="list-style-type: none"> • PWR_WKUP_COND_CALENDAR (Calendar Timer) • PWR_WKUP_COND_BOD_FEDGE (PMU Bod) • PWR_WKUP_COND_MSIO_COMP (Comparator) • PWR_WKUP_COND_ALL (all wakeup sources) |
| Return Value | None |
| Remarks | |

2.19.3.2 hal_pwr_config_timer_wakeup

Table 2-290 hal_pwr_config_timer_wakeup API

| | |
|-----------------------------|--|
| Function Prototype | void hal_pwr_config_timer_wakeup(uint8_t timer_mode, uint32_t load_count) |
| Function Description | Set the count value for AON SLEEP TIMER to wake up MCU from ultra deep sleep mode. |
| Parameter | <p>timer_mode: count mode for AON SLEEP TIMER. This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • PWR_SLP_TIMER_MODE_NORMAL (sleep mode; count in ultra deep sleep mode, and stop counting after the system is awoken) • PWR_SLP_TIMER_MODE_SINGLE (one-pulse count; continue counting after the system is awoken) • PWR_SLP_TIMER_MODE_RELOAD (automatic loading mode; continue counting after the system is awoken) • PWR_SLP_TIMER_MODE_DISABLE (disable) <p>load_count: time count to wake up MCU from ultra deep sleep mode; range: 0 to 0xFFFFFFFFU</p> |
| Return Value | None |
| Remarks | This API is available only when AON TIMER is included in the wakeup conditions. |

2.19.3.3 hal_pwr_config_ext_wakeup

Table 2-291 hal_pwr_config_ext_wakeup API

| | |
|-----------------------------|--|
| Function Prototype | void hal_pwr_config_ext_wakeup(uint32_t ext_wakeup_pinx, uint32_t ext_wakeup_type) |
| Function Description | Set AON_GPIO pins and types to wake up MCU from deep sleep mode. |
| Parameter | <p>ext_wakeup_pinx: AON_GPIO pins to wake up MCU. This parameter can be any combination of the following values:</p> <ul style="list-style-type: none"> • PWR_EXTWKUP_PIN0 (AON_GPIO Pin 0) • PWR_EXTWKUP_PIN1 (AON_GPIO Pin 1) • PWR_EXTWKUP_PIN2 (AON_GPIO Pin 2) • PWR_EXTWKUP_PIN3 (AON_GPIO Pin 3) • PWR_EXTWKUP_PIN4 (AON_GPIO Pin 4) |

| | |
|---------------------|---|
| | <ul style="list-style-type: none"> • PWR_EXTWKUP_PIN5 (AON_GPIO Pin 5) • PWR_EXTWKUP_PIN6 (AON_GPIO Pin 6) • PWR_EXTWKUP_PIN7 (AON_GPIO Pin 7) <p>ext_wakeup_type: AON_GPIO wakeup types to wake up MCU. This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • PWR_EXTWKUP_TYPE_LOW (triggered at low level) • PWR_EXTWKUP_TYPE_HIGH (triggered at high level) • PWR_EXTWKUP_TYPE_RISING (triggered by rising edge) • PWR_EXTWKUP_TYPE_FALLING (triggered by falling edge) |
| Return Value | None |
| Remarks | This API is available only when AON GPIO is included in the wakeup conditions. |

2.19.3.4 hal_pwr_set_comm_power

Table 2-292 hal_pwr_set_comm_power API

| | |
|-----------------------------|---|
| Function Prototype | void hal_pwr_set_comm_power(uint32_t timer_power_state, uint32_t core_power_state) |
| Function Description | Set power status for Bluetooth LE Core and Bluetooth LE Timer. |
| Parameter | <p>timer_power_state: power status for Bluetooth LE Timer. This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • PWR_COMM_TIMER_POWER_DOWN (Bluetooth LE Timer is in Power Down status.) • PWR_COMM_TIMER_POWER_UP (Bluetooth LE Timer is in Power Up status.) <p>core_power_state: power status for Bluetooth LE Core. This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • PWR_COMM_CORE_POWER_DOWN (Bluetooth LE Core is in Power Down status.) • PWR_COMM_CORE_POWER_UP (Bluetooth LE Core is in Power Up status.) |
| Return Value | None |
| Remarks | If Bluetooth LE functionalities are required, set Bluetooth LE Core and Bluetooth LE Timer as Power On status after booting MCU. |

2.19.3.5 hal_pwr_set_comm_mode

Table 2-293 hal_pwr_set_comm_mode API

| | |
|-----------------------------|---|
| Function Prototype | void hal_pwr_set_comm_mode(uint32_t timer_mode, uint32_t core_mode) |
| Function Description | Set the mode for Bluetooth LE Core and Bluetooth LE Timer. |
| Parameter | <p>timer_mode: the mode for Bluetooth LE Timer. This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • PWR_COMM_TIMER_MODE_RESET (Bluetooth LE Timer is in reset mode.) |

| | |
|---------------------|--|
| | <ul style="list-style-type: none"> • PWR_COMM_TIMER_MODE_RUNNING (Bluetooth LE Timer is in running mode.) <p>core_mode: the mode for Bluetooth LE Core. This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • PWR_COMM_CORE_MODE_RESET (Bluetooth LE Core is in reset mode.) • PWR_COMM_CORE_MODE_RUNNING (Bluetooth LE Core is in running mode.) |
| Return Value | None |
| Remarks | If Bluetooth LE functionalities are required, set Bluetooth LE Core and Bluetooth LE Timer as running mode after they are powered on. |

2.19.3.6 hal_pwr_enter_chip_deepsleep

Table 2-294 hal_pwr_enter_chip_deepsleep API

| | |
|-----------------------------|--|
| Function Prototype | void hal_pwr_enter_chip_deepsleep(void) |
| Function Description | Set MCU to enter the ultra deep sleep mode, and set memory blocks to be retained in ultra deep sleep mode and memory blocks to be powered full after being awoken. |
| Parameter | None |
| Return Value | None |
| Remarks | None |

2.19.3.7 hal_pwr_get_timer_current_value

Table 2-295 hal_pwr_get_timer_current_value API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_pwr_get_timer_current_value(uint32_t timer_type, uint32_t *p_value) |
| Function Description | Get the current timer value. |
| Parameter | <p>timer_type: timer type. This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • PWR_TIMER_TYPE_CAL_TIMER (CAL Timer) • PWR_TIMER_TYPE_AON_WDT (AON_WDT Timer) • PWR_TIMER_TYPE_SLP_TIMER (SLEEP Timer) • PWR_TIMER_TYPE_CAL_ALARM (CAL Alarm) <p>p_value: memory pointer. This parameter can be specified by developers.</p> |
| Return Value | HAL status |
| Remarks | Get the current timer value. |

2.19.3.8 hal_pwr_disable_ext_wakeup

Table 2-296 hal_pwr_disable_ext_wakeup API

| | |
|---------------------------|--|
| Function Prototype | void hal_pwr_disable_ext_wakeup(uint32_t disable_wakeup_pinx); |
|---------------------------|--|

| | |
|-----------------------------|--|
| Function Description | Disable the specified AON GPIO wakeup system. |
| Parameter | <p>disable_wakeup_pinx: a specified AON GPIO pin. This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • PWR_EXTWKUP_PIN0 • PWR_EXTWKUP_PIN1 • PWR_EXTWKUP_PIN2 • PWR_EXTWKUP_PIN3 • PWR_EXTWKUP_PIN4 • PWR_EXTWKUP_PIN5 • PWR_EXTWKUP_PIN6 • PWR_EXTWKUP_PIN7 • PWR_EXTWKUP_PIN_ALL |
| Return Value | None |
| Remarks | None |

2.19.3.9 hal_pwr_sleep_timer_irq_handler

Table 2-297 hal_pwr_sleep_timer_irq_handler API

| | |
|-----------------------------|--|
| Function Prototype | void hal_pwr_sleep_timer_irq_handler(void) |
| Function Description | Handle PWR Sleep Timer interrupt requests. |
| Parameter | None |
| Return Value | None |
| Remarks | |

2.19.3.10 hal_pwr_sleep_timer_elapsed_callback

Table 2-298 hal_pwr_sleep_timer_elapsed_callback API

| | |
|-----------------------------|--|
| Function Prototype | void hal_pwr_sleep_timer_elapsed_callback(void) |
| Function Description | Sleep Timer interrupt callback function |
| Parameter | None |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite this function before using it. |

2.20 HAL SPI Generic Driver

2.20.1 SPI Driver Functionalities

The HAL serial port interface (SPI) driver features the following functionalities:

- Motorola mode
- Master mode and slave mode supported, and a master device choosing two slave devices
- Up to 32 bits wide for data transfer
- Transfer rate at up to 32 MHz
- Configurable CPOL and CPHA
- Four operating modes: full duplex, simplex TX, simplex RX, and reading EEPROM.
- Setting and getting TX FIFO and RX FIFO thresholds.
- Three data read and write approaches: polling, interrupt, and DMA
- Aborting data read and write in interrupt/DMA mode
- Execution of interrupt callback functions at the end of TX, RX, TX and RX, and abort when errors occur
- Getting the operating states and error code of the driver
- Timeout settings

2.20.2 How to Use SPI Driver

Developers can:

1. Define a structure variable of `spi_handle_t`, such as `spi_handle_t spi_handle`.
2. Initialize the SPI low-level resources by overwriting `hal_spi_msp_init()`:
 - (1). Configure SPI pins for functionality multiplexing and enable pull-up resistors.
 - (2). Call relevant NVIC APIs to configure I/O APIs before using the interfaces in interrupt mode.
 - Configure the SPI interrupt priority by calling `hal_nvic_set_priority()`.
 - Enable SPI NVIC interrupts by calling `hal_nvic_enable_irq()`.
 - (3). Configure the DMA channels before using I/O APIs in DMA mode.
 - Define variables of `dma_handle_t` for TX/RX, such as `dma_handle_t dma_tx` and `dma_handle_t dma_rx`.
 - Configure parameters of DMA handle (`dma_tx` and `dma_rx`), for example, specifying TX or RX channels.
 - Point `p_dmatx` and `p_dmarx` (in `spi_handle`) to the initialized DMA handle variables `dma_tx` and `dma_rx`.
 - Configure the DMA interrupt priority, and enable NVIC interrupts for DMA.

3. Configure data transfer direction, data bit width, clock polarity, clock phase, baud rate prescaler values, and TI mode, and select the slave for SPI initialization structure.
4. Initialize SPI registers by calling `hal_spi_init(&spi_handle)`. The `hal_spi_init()` calls `hal_spi_msp_init(&spi_handle)` automatically to initialize SPI low-level resources.
5. HAL SPI driver provides three modes for SPI I/O operations (data read/write or memory read/write): polling, interrupt, and DMA.

2.20.3 SPI Driver Structures

2.20.3.1 spi_init_t

The initialization structure `spi_init_t` of SPI driver is defined below:

Table 2-299 spi_init_t structure

| Data Field | Field Description | Value |
|---------------------------------|--------------------|--|
| <code>uint32_t direction</code> | Transfer direction | This parameter can be one of the following values: <ul style="list-style-type: none"> • <code>SPI_DIRECTION_SIMPLEX_TX</code> (simplex TX) • <code>SPI_DIRECTION_SIMPLEX_RX</code> (simplex RX) • <code>SPI_DIRECTION_READ_EEPROM</code> (reading EEPROM) |
| <code>uint32_t data_size</code> | Data bit width | This parameter can be one of the following values: <ul style="list-style-type: none"> • <code>SPI_DATASIZE_4BIT</code> (4 bits) • <code>SPI_DATASIZE_5BIT</code> (5 bits) • <code>SPI_DATASIZE_6BIT</code> (6 bits) • <code>SPI_DATASIZE_7BIT</code> (7 bits) • <code>SPI_DATASIZE_8BIT</code> (8 bits) • <code>SPI_DATASIZE_9BIT</code> (9 bits) • <code>SPI_DATASIZE_10BIT</code> (10 bits) • <code>SPI_DATASIZE_11BIT</code> (11 bits) • <code>SPI_DATASIZE_12BIT</code> (12 bits) • <code>SPI_DATASIZE_13BIT</code> (13 bits) • <code>SPI_DATASIZE_14BIT</code> (14 bits) • <code>SPI_DATASIZE_15BIT</code> (15 bits) • <code>SPI_DATASIZE_16BIT</code> (16 bits) • <code>SPI_DATASIZE_17BIT</code> (17 bits) • <code>SPI_DATASIZE_18BIT</code> (18 bits) • <code>SPI_DATASIZE_19BIT</code> (19 bits) • <code>SPI_DATASIZE_20BIT</code> (20 bits) |

| Data Field | Field Description | Value |
|------------------------------|---------------------------|--|
| | | <ul style="list-style-type: none"> • SPI_DATASIZE_21BIT (21 bits) • SPI_DATASIZE_22BIT (22 bits) • SPI_DATASIZE_23BIT (23 bits) • SPI_DATASIZE_24BIT (24 bits) • SPI_DATASIZE_25BIT (25 bits) • SPI_DATASIZE_26BIT (26 bits) • SPI_DATASIZE_27BIT (27 bits) • SPI_DATASIZE_28BIT (28 bits) • SPI_DATASIZE_29BIT (29 bits) • SPI_DATASIZE_30BIT (30 bits) • SPI_DATASIZE_31BIT (31 bits) • SPI_DATASIZE_32BIT (32 bits) |
| uint32_t clk_polarity | Clock polarity | This parameter can be one of the following values: <ul style="list-style-type: none"> • SPI_POLARITY_LOW (clock idle at a low level) • SPI_POLARITY_HIGH (clock idle at a high level) |
| uint32_t clk_phase | Clock phase | This parameter can be one of the following values: <ul style="list-style-type: none"> • SPI_PHASE_1EDGE (data capture edge at the first clock transition) • SPI_PHASE_2EDGE (data capture edge at the second clock transition) |
| uint32_t baud_rate_prescaler | Baud rate prescaler value | Even numbers between 0x0000 and 0xFFFF. SPI transfer rate = System clock / prescaler value |
| uint32_t ti_mode | To enable TI mode or not | This parameter can be one of the following values: <ul style="list-style-type: none"> • SPI_TIMODE_DISABLE (TI mode disabled) • SPI_TIMODE_ENABLE (TI mode enabled) |
| uint32_t slave_select | Slave selection | This parameter can be one of the following values: <ul style="list-style-type: none"> • SPI_SLAVE_SELECT_0 (Slave 0) • SPI_SLAVE_SELECT_1 (Slave 1) • SPI_SLAVE_SELECT_ALL (Slave 0 and Slave 1) |

2.20.3.2 spi_handle_t

The handle structure of SPI driver spi_handle_t is defined below:

Table 2-300 spi_handle_t structure

| Data Field | Field Description | Value |
|--|---|---|
| ssi_regs_t *p_instance | SPI instance | This parameter can be one of the following values: <ul style="list-style-type: none"> • SPIM • SPIS |
| spi_init_t init | Initialization structure | See " Section 2.20.3.1 spi_init_t ". |
| uint8_t *p_tx_buffer | Pointer to data TX buffer (managed by SPI driver and initialization by developers not required) | N/A |
| __IO uint32_t tx_buffer_size | Data TX size (managed by SPI driver and initialization by developers not required) | N/A |
| __IO uint32_t tx_xfer_count | Data TX count (managed by SPI driver and initialization by developers not required) | N/A |
| uint8_t *p_rx_buffer | Pointer to data RX buffer (managed by SPI driver and initialization by developers not required) | N/A |
| __IO uint32_t rx_buffer_size | Data RX size (managed by SPI driver and initialization by developers not required) | N/A |
| __IO uint32_t rx_xfer_count | Data RX count (managed by SPI driver and initialization by developers not required) | N/A |
| void (*write_fifo)(struct _spi_handle *p_spi) | Pointer to the write FIFO function of SPI TX (managed by SPI driver and initialization by developers not required) | N/A |
| void (*read_fifo)(struct _spi_handle *p_spi) | Pointer to the read FIFO function of SPI TX (managed by SPI driver and initialization by developers not required) | N/A |
| void (*read_write_fifo)(struct _spi_handle *p_spi) | Pointer to the read and write FIFO function of SPI TX (managed by SPI driver and initialization by developers not required) | N/A |
| dma_handle_t *p_dmatx | Pointer in the DMA handle to data TX channels | DMA structure handler dma_handle_t of the data TX channel |
| dma_handle_t *p_dmarx | Pointer in the DMA handle to data RX channels | DMA handle structure dma_handle_t of the data RX channel |

| Data Field | Field Description | Value |
|----------------------------|--|---|
| __IO hal_lock_t lock | SPI lock (managed by SPI driver and initialization by developers not required) | N/A |
| __IO hal_spi_state_t state | SPI operating state (initialization by developers not required) | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • HAL_SPI_STATE_RESET (not initialized) • HAL_SPI_STATE_READY (initialized and ready for use) • HAL_SPI_STATE_BUSY (busy) • HAL_SPI_STATE_BUSY_TX (TX ongoing) • HAL_SPI_STATE_BUSY_RX (RX ongoing) • HAL_SPI_STATE_BUSY_TX_RX (TX and RX ongoing) • HAL_SPI_STATE_ABORT (aborted) • HAL_SPI_STATE_ERROR (error) |
| __IO uint32_t error_code | SPI error code (initialization by developers not required) | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • HAL_SPI_ERROR_NONE (no error) • HAL_SPI_ERROR_TIMEOUT (timeout) • HAL_SPI_ERROR_TRANSFER (transfer error) • HAL_SPI_ERROR_DMA (DMA transfer error) • HAL_SPI_ERROR_INVALID_PARAM (invalid parameter) |
| uint32_t timeout | SPI timeout period (initialization by developers not required) | N/A |
| uint32_t retention[8] | SPI register information (managed by SPI driver and initialization by developers not required) | N/A |

2.20.4 SPI Driver APIs

The SPI driver APIs are listed in the table below:

Table 2-301 SPI driver APIs

| API Type | API Name | Description |
|---------------------------------|---------------------------------|---|
| Initialization | hal_spi_init() | Initialize SPI, and configure clock prescaler values and other parameters. |
| | hal_spi_deinit() | Deinitialize SPI. |
| | hal_spi_msp_init() | Initialize GPIO pin multiplexing, NVIC interrupts, and DMA channels of SPI. |
| | hal_spi_msp_deinit() | Deinitialize GPIO pin multiplexing, NVIC interrupts, and DMA channels of SPI. |
| I/O operation | hal_spi_transmit() | Transmit data in polling mode. |
| | hal_spi_receive() | Receive data in polling mode. |
| | hal_spi_transmit_receive() | Transmit and receive data in polling mode. |
| | hal_spi_read_eeprom() | Read EEPROM in polling mode. |
| | hal_spi_transmit_it() | Transmit data in interrupt mode. |
| | hal_spi_receive_it() | Receive data in interrupt mode. |
| | hal_spi_transmit_receive_it() | Receive and transmit data in interrupt mode. |
| | hal_spi_read_eeprom_it() | Read EEPROM in interrupt mode. |
| | hal_spi_transmit_dma() | Transmit data in DMA mode. |
| | hal_spi_receive_dma() | Receive data in DMA mode. |
| | hal_spi_transmit_receive_dma() | Receive and transmit data in DMA mode. |
| | hal_spi_read_eeprom_dma() | Read EEPROM in DMA mode. |
| | hal_spi_abort() | In polling mode, abort data transfer in interrupt/DMA mode. |
| | hal_spi_abort_it() | In interrupt mode, abort data transfer in interrupt/DMA mode. |
| Interrupt handling and callback | hal_spi_irq_handler() | Interrupt handler |
| | hal_spi_tx_cplt_callback() | TX complete interrupt callback |
| | hal_spi_rx_cplt_callback() | RX complete interrupt callback |
| | hal_spi_tx_rx_cplt_callback() | RX and TX complete interrupt callback |
| | hal_spi_error_callback() | Error interrupt callback |
| | hal_spi_abort_cplt_callback() | Abort complete interrupt callback |
| State and error | hal_spi_get_state() | Get the driver operating state. |
| | hal_spi_get_error() | Get error code. |
| Control | hal_spi_set_timeout() | Set a timeout period. |
| | hal_spi_set_tx_fifo_threshold() | Set a TX FIFO threshold. |

| API Type | API Name | Description |
|----------|---------------------------------|---|
| | hal_spi_set_rx_fifo_threshold() | Set an RX FIFO threshold. |
| | hal_spi_get_tx_fifo_threshold() | Get a TX FIFO threshold. |
| | hal_spi_get_rx_fifo_threshold() | Get an RX FIFO threshold. |
| Sleep | hal_spi_suspend_reg() | Suspend registers related to SPI configuration in sleep mode. |
| | hal_spi_resume_reg() | Resume registers related to SPI configuration during wakeup. |

The sections below elaborate on these APIs.

2.20.4.1 hal_spi_init

Table 2-302 hal_spi_init API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_spi_init(spi_handle_t *p_spi) |
| Function Description | Initialize SPI and related handles according to parameters of " Section 2.20.3.1 spi_init_t ". |
| Parameter | p_spi: pointer to the variables of spi_handle_t . The variable contains the configuration information of a specified SPI. |
| Return Value | HAL status |
| Remarks | |

2.20.4.2 hal_spi_deinit

Table 2-303 hal_spi_deinit API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_spi_deinit(spi_handle_t *p_spi) |
| Function Description | Deinitialize SPI. |
| Parameter | p_spi: pointer to the variables of spi_handle_t . The variable contains the configuration information of a specified SPI. |
| Return Value | HAL status |
| Remarks | |

2.20.4.3 hal_spi_msp_init

Table 2-304 hal_spi_msp_init API

| | |
|-----------------------------|---|
| Function Prototype | void hal_spi_msp_init(spi_handle_t *p_spi) |
| Function Description | Initialize the GPIO pin multiplexing, NVIC interrupts, and DMA channels of SPI. |
| Parameter | p_spi: pointer to the variables of spi_handle_t . The variable contains the configuration information of a specified SPI. |

| | |
|---------------------|---|
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite the API to initialize GPIO pin multiplexing, NVIC interrupts, and DMA channels. |

2.20.4.4 hal_spi_msp_deinit

Table 2-305 hal_spi_msp_deinit API

| | |
|-----------------------------|---|
| Function Prototype | void hal_spi_msp_deinit(spi_handle_t *p_spi) |
| Function Description | Deinitialize the GPIO pin multiplexing, NVIC interrupts, and DMA channels of SPI. |
| Parameter | p_spi: pointer to the variables of spi_handle_t . The variable contains the configuration information of a specified SPI. |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite the API to deinitialize GPIO pin multiplexing, NVIC interrupts, and DMA channels. |

2.20.4.5 hal_spi_transmit

Table 2-306 hal_spi_transmit API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_spi_transmit(spi_handle_t *p_spi, uint8_t *p_data, uint32_t length, uint32_t timeout) |
| Function Description | Transmit a large volume of data in polling mode. |
| Parameter | p_spi: pointer to the variables of spi_handle_t . The variable contains the configuration information of a specified SPI. p_data: pointer to data buffer length: length of data to be transmitted timeout: timeout period |
| Return Value | HAL status |
| Remarks | When it returns HAL_ERROR, you can call hal_spi_get_error() to retrieve the error code. |

2.20.4.6 hal_spi_receive

Table 2-307 hal_spi_receive API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_spi_receive(spi_handle_t *p_spi, uint8_t *p_data, uint32_t length, uint32_t timeout) |
| Function Description | Receive a large amount of data in polling mode. |
| Parameter | p_spi: pointer to the variables of spi_handle_t . The variable contains the configuration information of a specified SPI. |

| | |
|---------------------|---|
| | <p>p_data: pointer to data buffer</p> <p>length: length of data to be received</p> <p>timeout: timeout period</p> |
| Return Value | HAL status |
| Remarks | When it returns HAL_ERROR, you can call hal_spi_get_error() to retrieve the error code. |

2.20.4.7 hal_spi_transmit_receive

Table 2-308 hal_spi_transmit_receive API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_spi_transmit_receive(spi_handle_t *p_spi, uint8_t *p_tx_data, uint8_t *p_rx_data, uint32_t length, uint32_t timeout) |
| Function Description | Transmit and receive a large amount of data in full-duplex and polling mode. |
| Parameter | <p>p_spi: pointer to the variables of spi_handle_t. The variable contains the configuration information of a specified SPI.</p> <p>p_tx_data: pointer to data TX buffer</p> <p>p_rx_data: pointer to data RX buffer</p> <p>length: length of data to be transmitted and received</p> <p>timeout: timeout period</p> |
| Return Value | HAL status |
| Remarks | When it returns HAL_ERROR, you can call hal_spi_get_error() to retrieve the error code. |

2.20.4.8 hal_spi_read_eeprom

Table 2-309 hal_spi_read_eeprom API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_spi_read_eeprom(spi_handle_t *p_spi, uint8_t *p_tx_data, uint8_t *p_rx_data, uint32_t tx_number_data, uint32_t rx_number_data, uint32_t timeout) |
| Function Description | Read data from EEPROM in half-duplex and polling mode. |
| Parameter | <p>p_spi: pointer to the variables of spi_handle_t. The variable contains the configuration information of a specified SPI.</p> <p>p_tx_data: pointer to data TX buffer</p> <p>p_rx_data: pointer to data RX buffer</p> <p>tx_number_data: length of data to be transmitted</p> <p>rx_number_data: length of data to be received</p> <p>timeout: timeout period</p> |
| Return Value | HAL status |
| Remarks | When it returns HAL_ERROR, you can call hal_spi_get_error() to retrieve the error code. |

2.20.4.9 hal_spi_transmit_it

Table 2-310 hal_spi_transmit_it API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_spi_transmit_it(spi_handle_t *p_spi, uint8_t *p_data, uint32_t length) |
| Function Description | Transmit a large volume of data in interrupt mode. |
| Parameter | <p>p_spi: pointer to the variables of spi_handle_t. The variable contains the configuration information of a specified SPI.</p> <p>p_data: pointer to data buffer</p> <p>length: length of data to be transmitted</p> |
| Return Value | HAL status |
| Remarks | <ul style="list-style-type: none"> When TX completes, the callback function hal_spi_tx_cplt_callback() will be called. When an error occurs during TX, the callback function hal_spi_error_callback() will be called. The related error code can be retrieved by calling hal_spi_get_error() in the callback function. Before calling hal_spi_tx_cplt_callback(), do not release the memory of the data buffer pointed by data. During transmission, if the SPI interrupt handler cannot respond in time, a data transmission error may occur. |

2.20.4.10 hal_spi_receive_it

Table 2-311 hal_spi_receive_it API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_spi_receive_it(spi_handle_t *p_spi, uint8_t *p_data, uint32_t length) |
| Function Description | Receive a large amount of data in interrupt mode. |
| Parameter | <p>p_spi: pointer to the variables of spi_handle_t. The variable contains the configuration information of a specified SPI.</p> <p>p_data: pointer to data buffer</p> <p>length: length of data to be received</p> |
| Return Value | HAL status |
| Remarks | <ul style="list-style-type: none"> When RX completes, the callback function hal_spi_rx_cplt_callback() will be called. When an error occurs during RX, the callback function hal_spi_error_callback() will be called. The related error code can be retrieved by calling hal_spi_get_error() in the callback function. Before calling hal_spi_rx_cplt_callback(), do not release the memory of the data buffer pointed by data. During transmission, if the SPI interrupt handler cannot respond in time, a data reception error may occur. |

2.20.4.11 hal_spi_transmit_receive_it

Table 2-312 hal_spi_transmit_receive_it API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_spi_transmit_receive_it(spi_handle_t *p_spi, uint8_t *p_tx_data, uint8_t *p_rx_data, uint32_t length) |
| Function Description | Transmit and receive a large amount of data in full-duplex and in interrupt mode. |
| Parameter | <p>p_spi: pointer to the variables of spi_handle_t. The variable contains the configuration information of a specified SPI.</p> <p>p_tx_data: pointer to data TX buffer</p> <p>p_rx_data: pointer to data RX buffer</p> <p>length: length of data to be transmitted and received</p> |
| Return Value | HAL status |
| Remarks | <ul style="list-style-type: none"> When TX and RX complete, the callback function hal_spi_tx_rx_cplt_callback() will be called. When an error occurs during RX/TX, the callback function hal_spi_error_callback() will be called. The related error code can be retrieved by calling hal_spi_get_error() in the callback function. Before calling hal_spi_tx_rx_cplt_callback(), do not release the memory of the data buffer pointed by data. During transmission, if the SPI interrupt handler cannot respond in time, a data transmission/reception error may occur. |

2.20.4.12 hal_spi_read_eeprom_it

Table 2-313 hal_spi_read_eeprom_it API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_spi_read_eeprom_it(spi_handle_t *p_spi, uint8_t *p_tx_data, uint8_t *p_rx_data, uint32_t tx_number_data, uint32_t rx_number_data) |
| Function Description | Read data from EEPROM in half-duplex and in interrupt mode. |
| Parameter | <p>p_spi: pointer to the variables of spi_handle_t. The variable contains the configuration information of a specified SPI.</p> <p>p_tx_data: pointer to data TX buffer</p> <p>p_rx_data: pointer to data RX buffer</p> <p>tx_number_data: length of data to be transmitted</p> <p>rx_number_data: length of data to be received</p> |
| Return Value | HAL status |
| Remarks | <ul style="list-style-type: none"> When read completes, the callback function hal_spi_rx_cplt_callback() will be called. When an error occurs during reading, the callback function hal_spi_error_callback() will be called. The related error code can be retrieved by calling hal_spi_get_error() in the callback function. Before calling hal_spi_rx_cplt_callback(), do not release the memory of the data buffer pointed by data. |

- During transmission, if the SPI interrupt handler cannot respond in time, a data read error may occur.

2.20.4.13 hal_spi_transmit_dma

Table 2-314 hal_spi_transmit_dma API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_spi_transmit_dma(spi_handle_t *p_spi, uint8_t *p_data, uint32_t length) |
| Function Description | Transmit a large amount of data in DMA mode. |
| Parameter | <p>p_spi: pointer to the variables of spi_handle_t. The variable contains the configuration information of a specified SPI.</p> <p>p_data: pointer to data buffer</p> <p>length: length of data to be transmitted</p> |
| Return Value | HAL status |
| Remarks | <ul style="list-style-type: none"> • When TX completes, the callback function hal_spi_tx_cplt_callback() will be called. • When an error occurs during TX, the callback function hal_spi_error_callback() will be called. The related error code can be retrieved by calling hal_spi_get_error() in the callback function. • Before calling hal_spi_tx_cplt_callback(), do not release the memory of the data buffer pointed by data. |

2.20.4.14 hal_spi_receive_dma

Table 2-315 hal_spi_receive_dma API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_spi_receive_dma(spi_handle_t *p_spi, uint8_t *p_data, uint32_t length) |
| Function Description | Receive a large amount of data in DMA mode. |
| Parameter | <p>p_spi: pointer to the variables of spi_handle_t. The variable contains the configuration information of a specified SPI.</p> <p>p_data: pointer to data buffer</p> <p>length: length of data to be received</p> |
| Return Value | HAL status |
| Remarks | <ul style="list-style-type: none"> • When RX completes, the callback function hal_spi_rx_cplt_callback() will be called. • When an error occurs during RX, the callback function hal_spi_error_callback() will be called. The related error code can be retrieved by calling hal_spi_get_error() in the callback function. • Before calling hal_spi_rx_cplt_callback(), do not release the memory of the data buffer pointed by data. |

2.20.4.15 hal_spi_transmit_receive_dma

Table 2-316 hal_spi_transmit_receive_dma API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_spi_transmit_receive_dma(spi_handle_t *p_spi, uint8_t *p_tx_data, uint8_t *p_rx_data, uint32_t length) |
| Function Description | Transmit and receive a large amount of data in full-duplex and in DMA mode. |
| Parameter | <p>p_spi: pointer to the variables of spi_handle_t. The variable contains the configuration information of a specified SPI.</p> <p>p_tx_data: pointer to data TX buffer</p> <p>p_rx_data: pointer to data RX buffer</p> <p>length: length of data to be transmitted and received</p> |
| Return Value | HAL status |
| Remarks | <ul style="list-style-type: none"> When TX and RX complete, the callback function hal_spi_tx_rx_cplt_callback() will be called. When an error occurs during RX/TX, the callback function hal_spi_error_callback() will be called. The related error code can be retrieved by calling hal_spi_get_error() in the callback function. Before calling hal_spi_tx_rx_cplt_callback(), do not release the memory of the data buffer pointed by data. |

2.20.4.16 hal_spi_read_eeprom_dma

Table 2-317 hal_spi_read_eeprom_dma API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_spi_read_eeprom_dma(spi_handle_t *p_spi, uint8_t *p_tx_data, uint8_t *p_rx_data, uint32_t tx_number_data, uint32_t rx_number_data) |
| Function Description | Read data from EEPROM in half-duplex and in DMA mode. |
| Parameter | <p>p_spi: pointer to the variables of spi_handle_t. The variable contains the configuration information of a specified SPI.</p> <p>p_tx_data: pointer to data TX buffer</p> <p>p_rx_data: pointer to data RX buffer</p> <p>tx_number_data: length of data to be transmitted</p> <p>rx_number_data: length of data to be received</p> |
| Return Value | HAL status |
| Remarks | <ul style="list-style-type: none"> When read completes, the callback function hal_spi_rx_cplt_callback() will be called. When an error occurs during reading, the callback function hal_spi_error_callback() will be called. The related error code can be retrieved by calling hal_spi_get_error() in the callback function. Before calling hal_spi_rx_cplt_callback(), do not release the memory of the data buffer pointed by data. |

2.20.4.17 hal_spi_abort

Table 2-318 hal_spi_abort API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_spi_abort(spi_handle_t *p_spi) |
| Function Description | In polling mode, abort data transfer in interrupt/DMA mode. |
| Parameter | p_spi: pointer to the variables of spi_handle_t . The variable contains the configuration information of a specified SPI. |
| Return Value | HAL status |
| Remarks | This is a polling function. It exits from the function when a TX completes. |

2.20.4.18 hal_spi_abort_it

Table 2-319 hal_spi_abort_it API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_spi_abort_it(spi_handle_t *p_spi) |
| Function Description | In interrupt mode, abort data transfer in interrupt/DMA mode. |
| Parameter | p_spi: pointer to the variables of spi_handle_t . The variable contains the configuration information of a specified SPI. |
| Return Value | HAL status |
| Remarks | This is a non-polling function. It exits from the function when TX and RX interrupts are enabled. After triggering TX_ABRT interrupt, abort completes, and hal_spi_abort_cplt_callback() will be called. |

2.20.4.19 hal_spi_irq_handler

Table 2-320 hal_spi_irq_handler API

| | |
|-----------------------------|---|
| Function Prototype | void hal_spi_irq_handler(spi_handle_t *p_spi) |
| Function Description | Handle SPI interrupt requests. |
| Parameter | p_spi: pointer to the variables of spi_handle_t . The variable contains the configuration information of a specified SPI. |
| Return Value | None |
| Remarks | |

2.20.4.20 hal_spi_tx_cplt_callback

Table 2-321 hal_spi_tx_cplt_callback API

| | |
|-----------------------------|---|
| Function Prototype | void hal_spi_tx_cplt_callback(spi_handle_t *p_spi) |
| Function Description | TX complete interrupt callback |
| Parameter | p_spi: pointer to the variables of spi_handle_t . The variable contains the configuration information of a specified SPI. |

| | |
|---------------------|--|
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite this function before using it. |

2.20.4.21 hal_spi_rx_cplt_callback

Table 2-322 hal_spi_rx_cplt_callback API

| | |
|-----------------------------|---|
| Function Prototype | void hal_spi_rx_cplt_callback(spi_handle_t *p_spi) |
| Function Description | RX complete interrupt callback |
| Parameter | p_spi: pointer to the variables of spi_handle_t . The variable contains the configuration information of a specified SPI. |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite this function before using it. |

2.20.4.22 hal_spi_tx_rx_cplt_callback

Table 2-323 hal_spi_tx_rx_cplt_callback API

| | |
|-----------------------------|---|
| Function Prototype | void hal_spi_tx_rx_cplt_callback(spi_handle_t *p_spi) |
| Function Description | RX and TX complete interrupt callback |
| Parameter | p_spi: pointer to the variables of spi_handle_t . The variable contains the configuration information of a specified SPI. |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite this function before using it. |

2.20.4.23 hal_spi_error_callback

Table 2-324 hal_spi_error_callback API

| | |
|-----------------------------|---|
| Function Prototype | void hal_spi_error_callback(spi_handle_t *p_spi) |
| Function Description | SPI error callback function |
| Parameter | p_spi: pointer to the variables of spi_handle_t . The variable contains the configuration information of a specified SPI. |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite this function before using it. |

2.20.4.24 hal_spi_abort_cplt_callback

Table 2-325 hal_spi_abort_cplt_callback API

| | |
|-----------------------------|---|
| Function Prototype | void hal_spi_abort_cplt_callback(spi_handle_t *p_spi) |
| Function Description | SPI abort complete callback function |
| Parameter | p_spi: pointer to the variables of spi_handle_t . The variable contains the configuration information of a specified SPI. |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite this function before using it. |

2.20.4.25 hal_spi_get_state

Table 2-326 hal_spi_get_state API

| | |
|-----------------------------|--|
| Function Prototype | hal_spi_state_t hal_spi_get_state(spi_handle_t *p_spi) |
| Function Description | Return the SPI operating state. |
| Parameter | p_spi: pointer to the variables of spi_handle_t . The variable contains the configuration information of a specified SPI. |
| Return Value | <p>SPI operating state can be one of the following values:</p> <ul style="list-style-type: none"> • HAL_SPI_STATE_RESET (not initialized) • HAL_SPI_STATE_READY (initialized and ready for use) • HAL_SPI_STATE_BUSY (busy) • HAL_SPI_STATE_BUSY_TX (TX ongoing) • HAL_SPI_STATE_BUSY_RX (RX ongoing) • HAL_SPI_STATE_BUSY_TX_RX (TX and RX ongoing) • HAL_SPI_STATE_ABORT (aborted) • HAL_SPI_STATE_ERROR (error) |
| Remarks | |

2.20.4.26 hal_spi_get_error

Table 2-327 hal_spi_get_error API

| | |
|-----------------------------|---|
| Function Prototype | uint32_t hal_spi_get_error(spi_handle_t *p_spi) |
| Function Description | Return the SPI error code. |
| Parameter | p_spi: pointer to the variables of spi_handle_t . The variable contains the configuration information of a specified SPI. |

| | |
|---------------------|--|
| Return Value | <p>SPI error code can be one of the following values:</p> <ul style="list-style-type: none"> • HAL_SPI_ERROR_NONE (no error) • HAL_SPI_ERROR_TIMEOUT (timeout) • HAL_SPI_ERROR_TRANSFER (transfer error) • HAL_SPI_ERROR_DMA (DMA transfer error) • HAL_SPI_ERROR_INVALID_PARAM (invalid parameter) |
| Remarks | |

2.20.4.27 hal_spi_set_timeout

Table 2-328 hal_spi_set_timeout API

| | |
|-----------------------------|---|
| Function Prototype | void hal_spi_set_timeout(spi_handle_t *p_spi, uint32_t timeout) |
| Function Description | Set timeout for SPI APIs. |
| Parameter | <p>p_spi: pointer to the variables of spi_handle_t. The variable contains the configuration information of a specified SPI.</p> <p>timeout: timeout period (ms)</p> |
| Return Value | None |
| Remarks | |

2.20.4.28 hal_spi_set_tx_fifo_threshold

Table 2-329 hal_spi_set_tx_fifo_threshold API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_spi_set_tx_fifo_threshold(spi_handle_t *p_spi, uint32_t threshold) |
| Function Description | Set a TX FIFO threshold. |
| Parameter | <p>p_spi: pointer to the variables of spi_handle_t. The variable contains the configuration information of a specified SPI.</p> <p>threshold: TX FIFO threshold</p> |
| Return Value | HAL status |
| Remarks | |

2.20.4.29 hal_spi_set_rx_fifo_threshold

Table 2-330 hal_spi_set_rx_fifo_threshold API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_spi_set_rx_fifo_threshold(spi_handle_t *p_spi, uint32_t threshold) |
| Function Description | Set an RX FIFO threshold. |
| Parameter | p_spi: pointer to the variables of spi_handle_t . The variable contains the configuration information of a specified SPI. |

| | |
|---------------------|------------------------------|
| | threshold: RX FIFO threshold |
| Return Value | HAL status |
| Remarks | |

2.20.4.30 hal_spi_get_tx_fifo_threshold

Table 2-331 hal_spi_get_tx_fifo_threshold API

| | |
|-----------------------------|---|
| Function Prototype | uint32_t hal_spi_get_tx_fifo_threshold(spi_handle_t *p_spi) |
| Function Description | Get a TX FIFO threshold. |
| Parameter | p_spi: pointer to the variables of spi_handle_t . The variable contains the configuration information of a specified SPI. |
| Return Value | TX FIFO threshold (value range: 0 to 7; 0 indicates that a TX FIFO is empty, and 7 indicates that a TX FIFO reaches 1 byte minus the full threshold.) |
| Remarks | |

2.20.4.31 hal_spi_get_rx_fifo_threshold

Table 2-332 hal_spi_get_rx_fifo_threshold API

| | |
|-----------------------------|---|
| Function Prototype | uint32_t hal_spi_get_rx_fifo_threshold(spi_handle_t *p_spi) |
| Function Description | Get an RX FIFO threshold. |
| Parameter | p_spi: pointer to the variables of spi_handle_t . The variable contains the configuration information of a specified SPI. |
| Return Value | RX FIFO threshold (value range: 0 to 7; 0 indicates that an RX FIFO is 1 byte, and 7 indicates that an RX FIFO is full.) |
| Remarks | |

2.20.4.32 hal_spi_suspend_reg

Table 2-333 hal_spi_suspend_reg API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_spi_suspend_reg(spi_handle_t *p_spi) |
| Function Description | Suspend registers related to SPI configuration in sleep mode. |
| Parameter | p_spi: pointer to the variables of spi_handle_t . The variable contains the configuration information of a specified SPI. |
| Return Value | HAL status |
| Remarks | |

2.20.4.33 hal_spi_resume_reg

Table 2-334 hal_spi_resume_reg API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_spi_resume_reg(spi_handle_t *p_spi) |
| Function Description | Resume registers related to SPI configuration during wakeup. |
| Parameter | p_spi: pointer to the variables of spi_handle_t . The variable contains the configuration information of a specified SPI. |
| Return Value | HAL status |
| Remarks | |

2.21 HAL TIMER Generic Driver

2.21.1 TIMER Driver Functionalities

The HAL TIMER driver features the following functionalities:

- Programmable 32-bit initial counting values
- Two counting approaches: polling and interrupt
- Stopping counting in polling/interrupt mode
- Counting complete interrupt callback function
- Getting the driver operating state

2.21.2 How to Use TIMER Driver

Developers can use TIMER driver in the following scenarios:

1. Declare a structure variable of `timer_handle_t`, for example: `timer_handle_t timer_handle`.
2. Initialize the TIMER low-level resources by overwriting `hal_timer_base_msp_init()`:
 - (1). If you count by using the interrupt API function `hal_timer_base_start_it()`, call the relevant NVIC APIs for configuration.
 - Configure the TIMER interrupt priority by calling `hal_nvic_set_priority()`.
 - Enable NVIC interrupt for TIMER by calling `hal_nvic_enable_irq()`.
 - (2). Configure the initial counting value in the init structure of `timer_handle`.
 - (3). Initialize TIMER peripheral by calling `hal_timer_base_init()` API.
3. If you count by running `hal_timer_base_start()` in polling mode, you can call `hal_timer_get_state()` to get the operating state of the driver, so as to check whether the current counting completes.
4. If you count by running `hal_timer_base_start_it()` in interrupt mode, you can overwrite the interrupt callback `hal_timer_period_elapsed_callback()`. When TIMER completes counting and interrupt is triggered, the callback function is called automatically.

2.21.3 TIMER Driver Structures

2.21.3.1 timer_init_t

The initialization structure timer_init_t of the TIMER driver is defined below:

Table 2-335 timer_init_t structure

| Data Field | Field Description | Value |
|----------------------|--------------------------------------|----------------------------|
| uint32_t auto_reload | Automatically reloaded initial count | 0x0000_0000 to 0xFFFF_FFFF |

2.21.3.2 timer_handle_t

The structure timer_handle_t of TIMER driver is defined below:

Table 2-336 timer_handle_t structure

| Data Field | Field Description | Value |
|------------------------------|---|--|
| timer_regs_t *p_instance | TIMER peripheral instance | This parameter can be one of the following values: <ul style="list-style-type: none"> TIMER 0 TIMER 1 |
| timer_init_t init | Initialization structure | See " Section 2.21.3.1 timer_init_t ". |
| __IO hal_lock_t lock | TIMER lock (initialization by developers not required) | N/A |
| __IO hal_timer_state_t state | TIMER operating state (initialization by developers not required) | This parameter can be one of the following values: <ul style="list-style-type: none"> HAL_TIMER_STATE_RESET HAL_TIMER_STATE_READY HAL_TIMER_STATE_BUSY HAL_TIMER_STATE_ERROR |

2.21.4 TIMER Driver APIs

The TIMER driver APIs are listed in the table below:

Table 2-337 TIMER driver APIs

| API Type | API Name | Description |
|----------------|-----------------------------|--|
| Initialization | hal_timer_base_init() | Initialize TIMER peripheral, and configure initial counting values and other parameters. |
| | hal_timer_base_deinit() | Deinitialize TIMER peripheral. |
| | hal_timer_base_msp_init() | Initialize NVIC interrupts of TIMER. |
| | hal_timer_base_msp_deinit() | Deinitialize NVIC interrupts of TIMER. |

| API Type | API Name | Description |
|---------------------------------|-------------------------------------|--|
| I/O operation | hal_timer_base_start() | Start counting in polling mode. |
| | hal_timer_base_stop() | Stop counting in polling mode. |
| | hal_timer_base_start_it() | Start counting in interrupt mode. |
| | hal_timer_base_stop_it() | Stop counting in interrupt mode. |
| Control | hal_timer_set_config() | Configure the TIMER. |
| Interrupt handling and callback | hal_timer_irq_handler() | Interrupt handler |
| | hal_timer_period_elapsed_callback() | Interrupt callback function at the end of counting |
| State and error | hal_timer_get_state() | Get the driver operating state. |

The sections below elaborate on these APIs.

2.21.4.1 hal_timer_base_init

Table 2-338 hal_timer_base_init API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_timer_base_init(timer_handle_t *p_timer) |
| Function Description | Initialize the TIMER time base unit and relevant handles based on specified parameters in timer_init_t . |
| Parameter | p_timer: pointer to variables of timer_handle_t . The variable contains the configuration information of a specified TIMER. |
| Return Value | HAL status |
| Remarks | |

2.21.4.2 hal_timer_base_deinit

Table 2-339 hal_timer_base_deinit API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_timer_base_deinit(timer_handle_t *p_timer) |
| Function Description | Deinitialize TIMER peripheral. |
| Parameter | p_timer: pointer to variables of timer_handle_t . The variable contains the configuration information of a specified TIMER. |
| Return Value | HAL status |
| Remarks | |

2.21.4.3 hal_timer_base_msp_init

Table 2-340 hal_timer_base_msp_init API

| | |
|-----------------------------|---|
| Function Prototype | void hal_timer_base_msp_init(timer_handle_t *p_timer) |
| Function Description | Initialize NVIC interrupts of TIMER. |

| | |
|---------------------|---|
| Parameter | p_timer: pointer to variables of timer_handle_t . The variable contains the configuration information of a specified TIMER. |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite the API to initialize NVIC interrupts. |

2.21.4.4 hal_timer_base_msp_deinit

Table 2-341 hal_timer_base_msp_deinit API

| | |
|-----------------------------|---|
| Function Prototype | void hal_timer_base_msp_deinit(timer_handle_t *p_timer) |
| Function Description | Deinitialize NVIC interrupts of TIMER. |
| Parameter | p_timer: pointer to variables of timer_handle_t . The variable contains the configuration information of a specified TIMER. |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite the API to deinitialize NVIC interrupts. |

2.21.4.5 hal_timer_base_start

Table 2-342 hal_timer_base_start API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_timer_base_start(timer_handle_t *p_timer) |
| Function Description | Enable TIMER and start counting in polling mode. |
| Parameter | p_timer: pointer to variables of timer_handle_t . The variable contains the configuration information of a specified TIMER. |
| Return Value | HAL status |
| Remarks | The API does not enable TIMER interrupts. Developers are required to call hal_timer_get_state() to get the counting state. |

2.21.4.6 hal_timer_base_stop

Table 2-343 hal_timer_base_stop API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_timer_base_stop(timer_handle_t *p_timer) |
| Function Description | Disable TIMER and stop counting in polling mode. |
| Parameter | p_timer: pointer to variables of timer_handle_t . The variable contains the configuration information of a specified TIMER. |
| Return Value | HAL status |

| | |
|----------------|--|
| Remarks | The API does not disable TIMER interrupts. Developers can run hal_timer_base_start() when calling the API. |
|----------------|--|

2.21.4.7 hal_timer_base_start_it

Table 2-344 hal_timer_base_start_it API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_timer_base_start_it(timer_handle_t *p_timer) |
| Function Description | Enable TIMER and start counting in interrupt mode. |
| Parameter | p_timer: pointer to variables of timer_handle_t . The variable contains the configuration information of a specified TIMER. |
| Return Value | HAL status |
| Remarks | The API enables TIMER interrupts. It calls hal_timer_period_elapsed_callback() when counting completes. |

2.21.4.8 hal_timer_base_stop_it

Table 2-345 hal_timer_base_stop_it API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_timer_base_stop_it(timer_handle_t *p_timer) |
| Function Description | Disable TIMER and stop counting in interrupt mode. |
| Parameter | p_timer: pointer to variables of timer_handle_t . The variable contains the configuration information of a specified TIMER. |
| Return Value | HAL status |
| Remarks | The API disables TIMER interrupts. Developers can run hal_timer_base_start_it() when calling the API. |

2.21.4.9 hal_timer_set_config

Table 2-346 hal_timer_set_config API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_timer_set_config(timer_handle_t *p_timer, timer_init_t *p_structure) |
| Function Description | Configure TIMER. |
| Parameter | p_timer: pointer to variables of timer_handle_t . The variable contains the configuration information of a specified TIMER. p_structure: pointer to variables of timer_init_t . The variable contains the parameters of a specified TIMER. |
| Return Value | HAL status |
| Remarks | The API is preferably called to reconfigure TIMER after initialization. |

2.21.4.10 hal_timer_irq_handler

Table 2-347 hal_timer_irq_handler API

| | |
|---------------------------|---|
| Function Prototype | void hal_timer_irq_handler(timer_handle_t *p_timer) |
|---------------------------|---|

| | |
|-----------------------------|---|
| Function Description | Handle TIMER interrupt requests. |
| Parameter | p_timer: pointer to variables of timer_handle_t . The variable contains the configuration information of a specified TIMER. |
| Return Value | None |
| Remarks | |

2.21.4.11 hal_timer_period_elapsed_callback

Table 2-348 hal_timer_period_elapsed_callback API

| | |
|-----------------------------|---|
| Function Prototype | void hal_timer_period_elapsed_callback(timer_handle_t *p_timer) |
| Function Description | Interrupt callback when TIMER counting completes |
| Parameter | p_timer: pointer to variables of timer_handle_t . The variable contains the configuration information of a specified TIMER. |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite this function before using it. |

2.21.4.12 hal_timer_get_state

Table 2-349 hal_timer_get_state API

| | |
|-----------------------------|---|
| Function Prototype | hal_timer_state_t hal_tim_get_state(timer_handle_t *p_timer) |
| Function Description | Get the TIMER operating state. |
| Parameter | p_timer: pointer to variables of timer_handle_t . The variable contains the configuration information of a specified TIMER. |
| Return Value | The TIMER operating state. This parameter can be one of the following values: <ul style="list-style-type: none"> • HAL_TIMER_STATE_RESET • HAL_TIMER_STATE_READY • HAL_TIMER_STATE_BUSY • HAL_TIMER_STATE_ERROR |
| Remarks | |

2.22 HAL Calendar Generic Driver

2.22.1 Calendar Driver Functionalities

The HAL Calendar driver features the following functionalities:

- 32-bit timer with a real-time clock (RTC, clock source: 32.768 kHz)

- Multiple prescaler values: 1, 32, 64, 128, and 256.
- Alarm
- Warp interrupt
- Setting date and time, and getting the current time

2.22.2 How to Use Calendar Driver

Developers can use the Calendar driver in the following scenarios:

1. Declare a structure variable of `timer_handle_t`, for example: `calendar_handle_t calendar_handle`.
2. Initialize Calendar peripheral by calling `hal_calendar_init()` API. Configure the initial time value in the init structure in `calendar_handle`, and launch Calendar in warp interrupt mode.
3. Call `hal_calendar_init_time()` and update Calendar time base according to information in `calendar_time_t`.
4. Call `hal_calendar_get_time()` to get the current Calendar time.
5. If you configure an alarm by calling `hal_calendar_set_alarm()`, you can overwrite the interrupt callback `hal_calendar_alarm_callback()`. When the Calendar completes counting and an alarm is triggered, the callback function is called automatically.
6. If you configure the timing of an alarm by millisecond by calling `hal_calendar_set_tick()`, you can overwrite the interrupt callback `hal_calendar_tick_callback()`. When the Calendar completes counting and an alarm is triggered, the callback function is called automatically.

2.22.3 Calendar Driver Structures

2.22.3.1 calendar_time_t

The `calendar_time_t` structure of Calendar driver is defined below:

Table 2-350 calendar_time_t structure

| Data Field | Field Description | Value |
|---------------------------|-------------------|---------|
| <code>uint8_t sec</code> | Second | 0 – 59 |
| <code>uint8_t min</code> | Minute | 0 – 59 |
| <code>uint8_t hour</code> | Hour | 0 – 23 |
| <code>uint8_t data</code> | Day | 1 – 31 |
| <code>uint8_t mon</code> | Month | 1 – 12 |
| <code>uint8_t year</code> | Year | 0 – 99 |
| <code>uint8_t week</code> | Week | 0 – 6 |
| <code>uint16_t ms</code> | Millisecond | 0 – 999 |

2.22.3.2 calendar_alarm_t

The `calendar_alarm_t` structure of Calendar driver is defined below:

Table 2-351 `calendar_alarm_t` structure

| Data Field | Field Description | Value |
|---|-------------------------|--|
| <code>uint8_t min</code> | Minute (Calendar alarm) | 0 – 59 |
| <code>uint8_t hour</code> | Hour (Calendar alarm) | 0 – 23 |
| <code>uint8_t alarm_sel</code> | Period (Calendar alarm) | This parameter can be one of the following values: <ul style="list-style-type: none"> • <code>CALENDAR_ALARM_SEL_DATE</code> • <code>CALENDAR_ALARM_SEL_WEEKDAY</code> |
| <code>uint8_t alarm_data_week_mask</code> | Date (Calendar alarm) | When <code>alarm_sel</code> is configured as <code>CALENDAR_ALARM_SEL_DATE</code> , the parameter ranges between 1 and 31. When <code>alarm_sel</code> is configured as <code>CALENDAR_ALARM_SEL_WEEKDAY</code> , the parameter can be one of the following values: <ul style="list-style-type: none"> • <code>CALENDAR_ALARM_WEEKDAY_SUN</code> • <code>CALENDAR_ALARM_WEEKDAY_MON</code> • <code>CALENDAR_ALARM_WEEKDAY_TUE</code> • <code>CALENDAR_ALARM_WEEKDAY_WED</code> • <code>CALENDAR_ALARM_WEEKDAY_THU</code> • <code>CALENDAR_ALARM_WEEKDAY_FRI</code> • <code>CALENDAR_ALARM_WEEKDAY_SAT</code> |

2.22.3.3 `calendar_handle_t`

The `calendar_handle_t` structure of Calendar driver is defined below:

Table 2-352 `calendar_handle_t` structure

| Data Field | Field Description | Value |
|--|--|---|
| <code>calendar_time_t time_init</code> | Calendar time structure | See " Section 2.22.3.1 <code>calendar_time_t</code> ". |
| <code>calendar_alarm_t alarm</code> | Calendar alarm structure | See " Section 2.22.3.2 <code>calendar_alarm_t</code> ". |
| <code>__IO hal_lock_t lock</code> | Calendar lock (initialization by developers not required) | N/A |
| <code>uint32_t prev_ms</code> | Accumulated count time of Calendar by millisecond | N/A |
| <code>uint32_t interval</code> | An alarm counting by millisecond | 5 to 3600000 (ms) |
| <code>uint8_t mode</code> | Alarm mode (initialization by developers not required) | N/A |
| <code>uint8_t sec</code> | Date alarm used to save the current time by second (initialization by developers not required) | N/A |
| <code>uint16_t ms</code> | Date alarm used to save the current time by millisecond (initialization by developers not required) | N/A |

2.22.4 Calendar Driver APIs

The Calendar driver APIs are listed in the table below:

Table 2-353 Calendar driver APIs

| API Type | API Name | Description |
|---------------------------------|-------------------------------|---|
| Initialization | hal_calendar_init() | Initialize Calendar and launch Calendar in warp interrupt mode. |
| | hal_calendar_deinit() | Deinitialize Calendar peripheral. |
| I/O operation | hal_calendar_init_time() | Initialize the current Calendar time. |
| | hal_calendar_get_time() | Get the current Calendar time. |
| | hal_calendar_set_alarm() | Set the time for Calendar alarms, and enable Calendar alarm. |
| | hal_calendar_set_tick() | Enable the Calendar alarm to count by millisecond, and enable Calendar alarm. |
| | hal_calendar_disable_event() | Disable Calendar alarm. |
| Interrupt handling and callback | hal_calendar_irq_handler() | Interrupt handler |
| | hal_calendar_alarm_callback() | Interrupt callback function of Calendar alarm. |
| | hal_calendar_tick_callback() | Interrupt callback function of Calendar alarm counting by millisecond |

The sections below elaborate on these APIs.

2.22.4.1 hal_calendar_init

Table 2-354 hal_calendar_init API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_calendar_init(calendar_handle_t *p_calendar) |
| Function Description | Initialize Calendar and launch Calendar in warp interrupt mode. |
| Parameter | p_calendar: pointer to the variables of calendar_handle_t . The variable contains the configuration information on a specified Calendar. |
| Return Value | HAL status |
| Remarks | |

2.22.4.2 hal_calendar_deinit

Table 2-355 hal_calendar_deinit API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_calendar_deinit(calendar_handle_t *p_calendar) |
| Function Description | Deinitialize Calendar peripheral. |

| | |
|---------------------|--|
| Parameter | p_calendar: pointer to the variables of calendar_handle_t . The variable contains the configuration information on a specified Calendar. |
| Return Value | HAL status |
| Remarks | |

2.22.4.3 hal_calendar_init_time

Table 2-356 hal_calendar_init_time API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_calendar_init_time(calendar_handle_t *p_calendar, calendar_time_t *p_time) |
| Function Description | Initialize the current Calendar time. |
| Parameter | p_calendar: pointer to the variables of calendar_handle_t . The variable contains the configuration information on a specified Calendar. p_time: pointer to the variables of calendar_time_t . The variable contains the configuration information on specified Calendar time. |
| Return Value | HAL status |
| Remarks | When this API is called, the minimum value of the year is 10, meaning Year 2010. |

2.22.4.4 hal_calendar_get_time

Table 2-357 hal_calendar_get_time API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_calendar_get_time(calendar_handle_t *p_calendar, calendar_time_t *p_time) |
| Function Description | Get the current Calendar time. |
| Parameter | p_calendar: pointer to the variables of calendar_handle_t . The variable contains the configuration information on a specified Calendar. p_time: pointer to variables of calendar_time_t . The variable contains information on the current Calendar time. |
| Return Value | HAL status |
| Remarks | |

2.22.4.5 hal_calendar_set_alarm

Table 2-358 hal_calendar_set_alarm API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_calendar_set_alarm(calendar_handle_t *p_calendar, calendar_alarm_t *p_alarm) |
| Function Description | Configure Calendar alarm based on p_alarm and enable Calendar alarm. |
| Parameter | p_calendar: pointer to the variables of calendar_handle_t . The variable contains the configuration information on a specified Calendar. p_alarm: pointer to the variables of calendar_alarm_t . The variable contains the configuration information on specified Calendar alarm time. |

| | |
|---------------------|--|
| Return Value | HAL status |
| Remarks | The API enables Calendar interrupt. When the counter reaches the alarm time, hal_calendar_alarm_callback() will be called. |

2.22.4.6 hal_calendar_set_tick

Table 2-359 hal_calendar_set_tick API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_hal_calendar_set_tick(calendar_handle_t *p_calendar, uint32_t interval) |
| Function Description | Set an alarm to count by millisecond. |
| Parameter | p_calendar: pointer to the variables of calendar_handle_t . The variable contains the configuration information on a specified Calendar. |
| Return Value | HAL status |
| Remarks | The minimum value is 5 ms, and the maximum value is 3600 x 1000 ms. |

2.22.4.7 hal_calendar_disable_event

Table 2-360 hal_calendar_disable_event API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_calendar_disable_event(calendar_handle_t *p_calendar, uint32_t disable_mode) |
| Function Description | Disable Calendar alarm. |
| Parameter | p_calendar: pointer to the variables of calendar_handle_t . The variable contains the configuration information on a specified Calendar. disable_mode: Choose the alarm to be disabled. The parameter can be configured as CALENDAR_ALARM_DISABLE_DATE, CALENDAR_ALARM_DISABLE_TICK, and CALENDAR_ALARM_DISABLE_ALL. |
| Return Value | HAL status |
| Remarks | The API disables Calendar alarm interrupts. Developers can run hal_calendar_set_alarm() when calling the API. |

2.22.4.8 hal_calendar_irq_handler

Table 2-361 hal_calendar_irq_handler API

| | |
|-----------------------------|--|
| Function Prototype | void hal_calendar_irq_handler(calendar_handle_t *p_calendar) |
| Function Description | Handle Calendar interrupt requests. |
| Parameter | p_calendar: pointer to the variables of calendar_handle_t . The variable contains the configuration information on a specified Calendar. |
| Return Value | None |
| Remarks | |

2.22.4.9 hal_calendar_alarm_callback

Table 2-362 hal_calendar_alarm_callback API

| | |
|-----------------------------|--|
| Function Prototype | void hal_calendar_alarm_callback(calendar_handle_t *p_calendar) |
| Function Description | Interrupt callback function of Calendar alarm. |
| Parameter | p_calendar: pointer to the variables of calendar_handle_t . The variable contains the configuration information on a specified Calendar. |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite this function before using it. |

2.22.4.10 hal_calendar_tick_callback

Table 2-363 hal_calendar_tick_callback API

| | |
|-----------------------------|--|
| Function Prototype | void hal_calendar_tick_callback(calendar_handle_t *p_calendar) |
| Function Description | Interrupt callback function of Calendar alarm counting by millisecond |
| Parameter | p_calendar: pointer to the variables of calendar_handle_t . The variable contains the configuration information on a specified Calendar. |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite this function before using it. |

2.23 HAL UART Generic Driver

2.23.1 UART Driver Functionalities

The HAL Universal Asynchronous Receiver/Transmitter (UART) driver features the following functionalities:

- Baud rate: 9600 bps to 921600 bps
- Data bit: 5, 6, 7, 8; odd-even parity check bits: no parity, odd parity, even parity, set to 0 or 1 (force); stop bit: 1, 1.5, 2
- Automatic flow control
- Three data read and write approaches: polling, interrupt, and DMA
- Aborting TX, RX, and both TX and RX in interrupt/DMA mode
- Suspension, resumption, and stop of TX and RX in DMA mode
- Interrupt callbacks for TX complete, RX complete, error, TX and RX abort complete, TX abort complete, RX abort complete
- Getting the operating states and error code of UART driver

2.23.2 How to Use UART Driver

Developers can use UART driver in the following scenarios:

1. Declare a structure variable of `uart_handle_t`, for example: `uart_handle_t uart_handle`.
2. Initialize the UART low-level resources by overwriting `hal_uart_msp_init()`:
 - (1). UART pin configuration: Configure the GPIO mode as `GPIO_MODE_MUX` by calling `hal_gpio_init()`, and configure the multiplexed functionalities of relevant GPIOs as UART.
 - (2). To use interrupt process (`hal_uart_transmit_it()` and `hal_uart_receive_it()` APIs), you need to make NVIC configurations:
 - Configure the UART interrupt priority by calling `hal_nvic_set_priority()`.
 - Enable NVIC interrupts for UART by calling `hal_nvic_enable_irq()`.
 - (3). To use DMA process (`hal_uart_transmit_DMA()` and `hal_uart_receive_DMA()` APIs), you need to configure DMA:
 - Declare a DMA handle structure for TX/RX channels, for example: `dma_handle_t htxdma`.
 - Configure the declared DMA handle structure by using the required TX/RX parameters.
 - Configure DMA TX/RX channels.
 - Associate the initialized DMA handle with UART DMA TX/RX handles.
 - Configure the priority and enable the NVIC for transfer complete interrupt on DMA TX/RX channels.
3. Configure the baud rate, data bit, stop bit, parity bit, hardware flow control, and mode (receiver/transmitter) in the init structure of `uart_handle`.
4. Initialize UART registers by calling `hal_uart_init()`.

2.23.3 UART Driver Structures

2.23.3.1 `uart_init_t`

The initialization structure `timer_init_t` of UART driver is defined below:

Table 2-364 `uart_init_t` structure

| Data Field | Field Description | Value |
|---------------------------------|-------------------|---|
| <code>uint32_t baud_rate</code> | Baud rate | 9600 – 921600 |
| <code>uint32_t data_bits</code> | Data bit | This parameter can be one of the following values: <ul style="list-style-type: none"> • <code>UART_DATABITS_5B</code> (5 bits) • <code>UART_DATASIZE_6B</code> (6 bits) • <code>UART_DATABITS_7B</code> (7 bits) • <code>UART_DATABITS_8B</code> (8 bits) |

| Data Field | Field Description | Value |
|--------------------------|-----------------------|--|
| uint32_t stop_bits | Stop bit | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> UART_STOPBITS_1 (1 bit) UART_STOPBITS_1_5 (1.5 bits) UART_STOPBITS_2 (2 bits) <p>The stop bit can be UART_STOPBITS_1_5 only when the data bit is UART_DATABITS_5B; the stop bit parameter cannot be UART_STOPBITS_2 when the data bit is UART_DATABITS_5B.</p> |
| uint32_t parity | Parity bit | <p>This parameter can be one of the following values:</p> <p>UART_PARITY_NONE (no parity)</p> <p>UART_PARITY_EVEN (even parity)</p> <p>UART_PARITY_ODD (odd parity)</p> <p>UART_PARITY_SPO (parity bit = 0)</p> <p>UART_PARITY_SP1 (parity bit = 1)</p> |
| uint32_t hw_flow_ctrl | Hardware flow control | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> UART_HWCONTROL_NONE (no hardware control) UART_HWCONTROL_RTS_CTS (automatic flow control) |
| uint32_t rx_timeout_mode | Receiver timeout | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> UART_RECEIVER_TIMEOUT_DISABLE (UART receiver timeout disabled) UART_RECEIVER_TIMEOUT_ENABLE (UART receiver timeout enabled) |

2.23.3.2 uart_handle_t

The handle structure `uart_handle_t` of UART driver is defined below:

Table 2-365 `uart_handle_t` structure

| Data Field | Field Description | Value |
|-------------------------|--|--|
| uart_regs_t *p_instance | UART peripheral instance | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> UART 0 UART 1 |
| uart_init_t init | Initialization structure | See " Section 2.23.3.1 uart_init_t ". |
| uint8_t *p_tx_buffer | Pointer to data TX buffer (managed by UART driver and initialization by developers not required) | N/A |

| Data Field | Field Description | Value |
|--------------------------------|--|--|
| __IO uint16_t tx_xfer_size | Data TX size (managed by UART driver and initialization by developers not required) | N/A |
| __IO uint16_t tx_xfer_count | Data TX count (managed by UART driver and initialization by developers not required) | N/A |
| uint8_t *p_rx_buffer | Pointer to data RX buffer (managed by UART driver and initialization by developers not required) | N/A |
| __IO uint16_t rx_xfer_size | Data RX size (managed by UART driver and initialization by developers not required) | N/A |
| __IO uint16_t rx_xfer_count | Data RX count (managed by UART driver and initialization by developers not required) | N/A |
| dma_handle_t *p_dmatx | Pointer in the DMA handle to data TX channels (managed by UART driver and initialization by developers not required) | N/A |
| dma_handle_t *p_dmarx | Pointer in the DMA handle to data RX channels | DMA handle structure dma_handle_t of the data RX channel |
| __IO hal_lock_t lock | UART lock (managed by UART driver and initialization by developers not required) | N/A |
| __IO hal_uart_state_t tx_state | UART operating state (initialization by developers not required) | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • HAL_UART_STATE_RESET (not initialized) • HAL_UART_STATE_READY (initialized and ready for use) • HAL_UART_STATE_BUSY (busy) • HAL_UART_STATE_BUSY_TX (TX ongoing) • HAL_UART_STATE_BUSY_RX (RX ongoing) • HAL_UART_STATE_TIMEOUT (timeout) • HAL_UART_STATE_ERROR (error) |
| __IO hal_uart_state_t rx_state | UART RX state (initialization by developers not required) | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • HAL_UART_STATE_RESET (not initialized) |

| Data Field | Field Description | Value |
|---------------------------|--|---|
| | | <ul style="list-style-type: none"> • HAL_UART_STATE_READY (initialized and ready for use) • HAL_UART_STATE_BUSY (busy) • HAL_UART_STATE_BUSY_TX (TX ongoing) • HAL_UART_STATE_BUSY_RX (RX ongoing) • HAL_UART_STATE_TIMEOUT (timeout) • HAL_UART_STATE_ERROR (error) |
| __IO hal_uart_mode_t mode | UART operating mode (managed by UART driver and initialization by developers not required) | N/A |
| __IO uint32_t error_code | UART error code (initialization by developers not required) | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • HAL_UART_ERROR_NONE (no error) • HAL_UART_ERROR_PE (parity error) • HAL_UART_ERROR_FE (frame error) • HAL_UART_ERROR_OE (overflow error) • HAL_UART_ERROR_BI (line breaking error) • HAL_UART_ERROR_DMA (DMA transfer error) • HAL_UART_ERROR_BUSY (busy) |
| uint32_t retention[8] | UART register information (managed by UART driver and initialization by developers not required) | N/A |

2.23.4 UART Driver APIs

The UART driver APIs are listed in the table below:

Table 2-366 UART driver APIs

| API Type | API Name | Description |
|----------------|-----------------------|--|
| Initialization | hal_uart_init() | Initialize UART peripheral, and configure clock prescaler values and other parameters. |
| | hal_uart_deinit() | Deinitialize UART peripheral. |
| | hal_uart_msp_init() | Initialize GPIO pin multiplexing, NVIC interrupts, and DMA channels of UART. |
| | hal_uart_msp_deinit() | Deinitialize GPIO pin multiplexing, NVIC interrupts, and DMA channels of UART. |
| I/O operation | hal_uart_transmit() | Transmit data in polling mode. |

| API Type | API Name | Description |
|---------------------------------|---|--|
| | hal_uart_receive() | Receive data in polling mode. |
| | hal_uart_transmit_it() | Transmit data in interrupt mode. |
| | hal_uart_receive_it() | Receive data in interrupt mode. |
| | hal_uart_transmit_dma() | Transmit data in DMA mode. |
| | hal_uart_receive_dma() | Receive data in DMA mode. |
| | hal_uart_dma_pause() | Pause DMA transfer. |
| | hal_uart_dma_resume() | Resume DMA transfer. |
| | hal_uart_dma_stop() | Stop DMA transfer. |
| | hal_uart_abort() | In polling mode, abort data TX and RX in interrupt/DMA mode. |
| | hal_uart_abort_transmit() | In polling mode, abort data TX in interrupt/DMA mode. |
| | hal_uart_abort_receive() | In polling mode, abort data RX in interrupt/DMA mode. |
| | hal_uart_abort_it() | In interrupt mode, abort data TX and RX in interrupt/DMA mode. |
| | hal_uart_abort_transmit_it() | In interrupt mode, abort data TX in interrupt/DMA mode. |
| | hal_uart_abort_receive_it() | In interrupt mode, abort data RX in interrupt/DMA mode. |
| Interrupt handling and callback | hal_uart_irq_handler() | Interrupt handler |
| | hal_uart_tx_cplt_callback() | TX complete interrupt callback |
| | hal_uart_rx_cplt_callback() | RX complete interrupt callback |
| | hal_uart_error_callback() | Error interrupt callback |
| | hal_uart_abort_cplt_callback() | TX and RX abort complete interrupt callback function |
| | hal_uart_abort_transmit_cplt_callback() | TX abort complete interrupt callback function |
| | hal_uart_abort_receive_cplt_callback() | RX abort complete interrupt callback function |
| State and error | hal_uart_get_state() | Get the driver operating state. |
| | hal_uart_get_error() | Get error code. |
| Sleep | hal_uart_suspend_reg() | Suspend registers related to UART configuration before the device enters sleep mode. |
| | hal_uart_resume_reg() | Resume registers related to UART configuration during wakeup. |

The sections below elaborate on these APIs.

2.23.4.1 hal_uart_init

Table 2-367 hal_uart_init API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_uart_init(uart_handle_t *p_uart) |
| Function Description | Initialize UART mode and relevant handles based on specified parameters in uart_init_t . |

| | |
|---------------------|--|
| Parameter | p_uart: pointer to the variables of uart_handle_t . The variable contains the configuration information on a specified UART. |
| Return Value | HAL status |
| Remarks | |

2.23.4.2 hal_uart_deinit

Table 2-368 hal_uart_deinit API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_uart_deinit(uart_handle_t *p_uart) |
| Function Description | Deinitialize UART peripheral. |
| Parameter | p_uart: pointer to the variables of uart_handle_t . The variable contains the configuration information on a specified UART. |
| Return Value | HAL status |
| Remarks | |

2.23.4.3 hal_uart_msp_init

Table 2-369 hal_uart_msp_init API

| | |
|-----------------------------|--|
| Function Prototype | void hal_uart_msp_init(uart_handle_t *p_uart) |
| Function Description | Initialize the multiplexed GPIO pins, NVIC interrupts, and DMA channels of UART. |
| Parameter | p_uart: pointer to the variables of uart_handle_t . The variable contains the configuration information on a specified UART. |
| Return Value | None |
| Remarks | The API is declared empty as weak function. Developers are required to overwrite the API to initialize the multiplexed GPIO pins, NVIC interrupts, and DMA channels. |

2.23.4.4 hal_uart_msp_deinit

Table 2-370 hal_uart_msp_deinit API

| | |
|-----------------------------|--|
| Function Prototype | void hal_uart_msp_deinit(uart_handle_t *p_uart) |
| Function Description | Deinitialize the multiplexed GPIO pins, NVIC interrupts, and DMA channels of UART. |
| Parameter | p_uart: pointer to the variables of uart_handle_t . The variable contains the configuration information on a specified UART. |
| Return Value | None |
| Remarks | The API is declared empty as weak function. Developers are required to overwrite the API to deinitialize GPIO pin multiplexing, NVIC interrupts, and DMA channels. |

2.23.4.5 hal_uart_transmit

Table 2-371 hal_uart_transmit API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_uart_transmit(uart_handle_t *p_uart, uint8_t *p_data, uint16_t size, uint32_t timeout) |
| Function Description | Transmit a large volume of data in polling mode. |
| Parameter | <p>p_uart: pointer to the variables of uart_handle_t. The variable contains the configuration information on a specified UART.</p> <p>p_data: pointer to data buffer</p> <p>size: size of data to be transmitted</p> <p>timeout: timeout period</p> |
| Return Value | HAL status |
| Remarks | |

2.23.4.6 hal_uart_receive

Table 2-372 hal_uart_receive API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_uart_receive(uart_handle_t *p_uart, uint8_t *p_data, uint16_t size, uint32_t timeout) |
| Function Description | Receive a large amount of data in polling mode. |
| Parameter | <p>p_uart: pointer to the variables of uart_handle_t. The variable contains the configuration information on a specified UART.</p> <p>p_data: pointer to data buffer</p> <p>size: size of data to be received</p> <p>timeout: timeout period</p> |
| Return Value | HAL status |
| Remarks | |

2.23.4.7 hal_uart_transmit_it

Table 2-373 hal_uart_transmit_it API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_uart_transmit_it(uart_handle_t *p_uart, uint8_t *p_data, uint16_t size) |
| Function Description | Transmit a large volume of data in interrupt mode. |
| Parameter | <p>p_uart: pointer to the variables of uart_handle_t. The variable contains the configuration information on a specified UART.</p> <p>p_data: pointer to data buffer</p> <p>size: size of data to be transmitted</p> |
| Return Value | HAL status |
| Remarks | |

2.23.4.8 hal_uart_receive_it

Table 2-374 hal_uart_receive_it API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_uart_receive_it(uart_handle_t *p_uart, uint8_t *p_data, uint16_t size) |
| Function Description | Receive a large amount of data in interrupt mode. |
| Parameter | p_uart: pointer to the variables of uart_handle_t . The variable contains the configuration information on a specified UART. p_data: pointer to data buffer size: size of data to be received |
| Return Value | HAL status |
| Remarks | |

2.23.4.9 hal_uart_transmit_dma

Table 2-375 hal_uart_transmit_dma API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_uart_transmit_dma(uart_handle_t *p_uart, uint8_t *p_data, uint16_t size) |
| Function Description | Transmit a large amount of data in DMA mode. |
| Parameter | p_uart: pointer to the variables of uart_handle_t . The variable contains the configuration information on a specified UART. p_data: pointer to data buffer size: size of data to be transmitted |
| Return Value | HAL status |
| Remarks | |

2.23.4.10 hal_uart_receive_dma

Table 2-376 hal_uart_receive_dma API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_uart_receive_dma(uart_handle_t *p_uart, uint8_t *p_data, uint16_t size) |
| Function Description | Receive a large amount of data in DMA mode. |
| Parameter | p_uart: pointer to the variables of uart_handle_t . The variable contains the configuration information on a specified UART. p_data: pointer to data buffer size: size of data to be received |
| Return Value | HAL status |
| Remarks | |

2.23.4.11 hal_uart_dma_pause

Table 2-377 hal_uart_dma_pause API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_uart_dma_pause(uart_handle_t *p_uart) |
| Function Description | Pause UART DMA transfer. |
| Parameter | p_uart: pointer to the variables of uart_handle_t . The variable contains the configuration information on a specified UART. |
| Return Value | HAL status |
| Remarks | |

2.23.4.12 hal_uart_dma_resume

Table 2-378 hal_uart_dma_resume API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_uart_dma_resume(uart_handle_t *p_uart) |
| Function Description | Resume UART DMA transfer. |
| Parameter | p_uart: pointer to the variables of uart_handle_t . The variable contains the configuration information on a specified UART. |
| Return Value | HAL status |
| Remarks | |

2.23.4.13 hal_uart_dma_stop

Table 2-379 hal_uart_dma_stop API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_uart_dma_stop(uart_handle_t *p_uart) |
| Function Description | Stop UART DMA transfer. |
| Parameter | p_uart: pointer to the variables of uart_handle_t . The variable contains the configuration information on a specified UART. |
| Return Value | HAL status |
| Remarks | |

2.23.4.14 hal_uart_abort

Table 2-380 hal_uart_abort API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_uart_abort(uart_handle_t *p_uart) |
| Function Description | In polling mode, abort data TX and RX in interrupt/DMA mode. |
| Parameter | p_uart: pointer to the variables of uart_handle_t . The variable contains the configuration information on a specified UART. |
| Return Value | HAL status |

| | |
|----------------|--|
| Remarks | <p>The function can be applied to abort data TX and RX in interrupt/DMA mode. The function can:</p> <ul style="list-style-type: none"> • Disable TX interrupt and RX interrupt. • Stop DMA transfer. • Set the state in p_uart as HAL_UART_STATE_READY. |
|----------------|--|

2.23.4.15 hal_uart_abort_transmit

Table 2-381 hal_uart_abort_transmit API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_uart_abort_transmit(uart_handle_t *p_uart) |
| Function Description | In polling mode, abort data TX in interrupt/DMA mode. |
| Parameter | p_uart: pointer to the variables of uart_handle_t . The variable contains the configuration information on a specified UART. |
| Return Value | HAL status |
| Remarks | <p>The function can be applied to abort data TX in interrupt/DMA mode. The function can:</p> <ul style="list-style-type: none"> • Disable TX interrupt. • Stop DMA transfer. • Set the state in p_uart as HAL_UART_STATE_READY. |

2.23.4.16 hal_uart_abort_receive

Table 2-382 hal_uart_abort_receive API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_uart_abort_receive(uart_handle_t *p_uart) |
| Function Description | In polling mode, abort data RX in interrupt/DMA mode. |
| Parameter | p_uart: pointer to the variables of uart_handle_t . The variable contains the configuration information on a specified UART. |
| Return Value | HAL status |
| Remarks | <p>This function can be applied to abort data RX in interrupt/DMA mode. The function can:</p> <ul style="list-style-type: none"> • Disable RX interrupt. • Stop DMA transfer. • Set the state in p_uart as HAL_UART_STATE_READY. |

2.23.4.17 hal_uart_abort_it

Table 2-383 hal_uart_abort_it API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_uart_abort_it(uart_handle_t *p_uart) |
| Function Description | In interrupt mode, abort data TX and RX in interrupt/DMA mode. |

| | |
|---------------------|--|
| Parameter | p_uart: pointer to the variables of uart_handle_t . The variable contains the configuration information on a specified UART. |
| Return Value | HAL status |
| Remarks | <p>The function can be applied to abort data TX and RX in interrupt/DMA mode. The function can:</p> <ul style="list-style-type: none"> • Disable TX interrupt and RX interrupt. • Stop DMA transfer. • Set the state in p_uart as HAL_UART_STATE_READY. <p>After abort completes, hal_uart_abort_cplt_callback() is called.</p> |

2.23.4.18 hal_uart_abort_transmit_it

Table 2-384 hal_uart_abort_transmit_it API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_uart_abort_transmit_it(uart_handle_t *p_uart) |
| Function Description | In interrupt mode, abort data TX in interrupt/DMA mode. |
| Parameter | p_uart: pointer to the variables of uart_handle_t . The variable contains the configuration information on a specified UART. |
| Return Value | HAL status |
| Remarks | <p>The function can be applied to abort data TX in interrupt/DMA mode. The function can:</p> <ul style="list-style-type: none"> • Disable TX interrupt. • Stop DMA transfer. • Set the state in p_uart as HAL_UART_STATE_READY. <p>After abort completes, hal_uart_abort_transmit_cplt_callback() is called.</p> |

2.23.4.19 hal_uart_abort_receive_it

Table 2-385 hal_uart_abort_receive_it API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_uart_abort_receive_it(uart_handle_t *p_uart) |
| Function Description | In interrupt mode, abort data RX in interrupt/DMA mode. |
| Parameter | p_uart: pointer to the variables of uart_handle_t . The variable contains the configuration information on a specified UART. |
| Return Value | HAL status |
| Remarks | <p>This function can be applied to abort data RX in interrupt/DMA mode. The function can:</p> <ul style="list-style-type: none"> • Disable RX interrupt. • Stop DMA transfer. • Set the state in p_uart as HAL_UART_STATE_READY. |

After abort completes, `hal_uart_abort_receive_cplt_callback()` is called.

2.23.4.20 hal_uart_irq_handler

Table 2-386 hal_uart_irq_handler API

| | |
|-----------------------------|---|
| Function Prototype | void hal_uart_irq_handler(uart_handle_t *p_uart) |
| Function Description | Handle UART interrupt requests. |
| Parameter | p_uart: pointer to the variables of <code>uart_handle_t</code> . The variable contains the configuration information on a specified UART. |
| Return Value | None |
| Remarks | |

2.23.4.21 hal_uart_tx_cplt_callback

Table 2-387 hal_uart_tx_cplt_callback API

| | |
|-----------------------------|---|
| Function Prototype | void hal_uart_tx_cplt_callback(uart_handle_t *p_uart) |
| Function Description | TX complete interrupt callback |
| Parameter | p_uart: pointer to the variables of <code>uart_handle_t</code> . The variable contains the configuration information on a specified UART. |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite this function before using it. |

2.23.4.22 hal_uart_rx_cplt_callback

Table 2-388 hal_uart_rx_cplt_callback API

| | |
|-----------------------------|---|
| Function Prototype | void hal_uart_rx_cplt_callback(uart_handle_t *p_uart) |
| Function Description | RX complete interrupt callback |
| Parameter | p_uart: pointer to the variables of <code>uart_handle_t</code> . The variable contains the configuration information on a specified UART. |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite this function before using it. |

2.23.4.23 hal_uart_error_callback

Table 2-389 hal_uart_error_callback API

| | |
|-----------------------------|--|
| Function Prototype | void hal_uart_error_callback(uart_handle_t *p_uart) |
| Function Description | UART error interrupt callback function |
| Parameter | p_uart: pointer to the variables of uart_handle_t . The variable contains the configuration information on a specified UART. |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite this function before using it. |

2.23.4.24 hal_uart_abort_cplt_callback

Table 2-390 hal_uart_abort_cplt_callback API

| | |
|-----------------------------|--|
| Function Prototype | void hal_uart_abort_cplt_callback(uart_handle_t *p_uart) |
| Function Description | UART abort complete interrupt callback |
| Parameter | p_uart: pointer to the variables of uart_handle_t . The variable contains the configuration information on a specified UART. |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite this function before using it. |

2.23.4.25 hal_uart_abort_tx_cplt_callback

Table 2-391 hal_uart_abort_tx_cplt_callback API

| | |
|-----------------------------|--|
| Function Prototype | void hal_uart_abort_tx_cplt_callback(uart_handle_t *p_uart) |
| Function Description | UART abort TX complete interrupt callback function |
| Parameter | p_uart: pointer to the variables of uart_handle_t . The variable contains the configuration information on a specified UART. |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite this function before using it. |

2.23.4.26 hal_uart_abort_rx_cplt_callback

Table 2-392 hal_uart_abort_rx_cplt_callback API

| | |
|-----------------------------|---|
| Function Prototype | void hal_uart_abort_rx_cplt_callback(uart_handle_t *p_uart) |
| Function Description | UART abort RX complete interrupt callback function |

| | |
|---------------------|--|
| Parameter | p_uart: pointer to the variables of uart_handle_t . The variable contains the configuration information on a specified UART. |
| Return Value | None |
| Remarks | |

2.23.4.27 hal_uart_get_state

Table 2-393 hal_uart_get_state API

| | |
|-----------------------------|--|
| Function Prototype | hal_uart_state_t hal_uart_get_state(uart_handle_t *p_uart) |
| Function Description | Get UART operating state. |
| Parameter | p_uart: pointer to the variables of uart_handle_t . The variable contains the configuration information on a specified UART. |
| Return Value | <p>UART operating state can be one of the following values:</p> <ul style="list-style-type: none"> • HAL_UART_STATE_RESET (not initialized) • HAL_UART_STATE_READY (initialized and ready for use) • HAL_UART_STATE_BUSY (busy) • HAL_UART_STATE_BUSY_TX (TX ongoing) • HAL_UART_STATE_BUSY_RX (RX ongoing) • HAL_UART_STATE_TIMEOUT (timeout) • HAL_UART_STATE_ERROR (error) |
| Remarks | |

2.23.4.28 hal_uart_get_error

Table 2-394 hal_uart_get_error API

| | |
|-----------------------------|---|
| Function Prototype | uint32_t hal_uart_get_error(uart_handle_t *p_uart) |
| Function Description | Get UART error code. |
| Parameter | p_uart: pointer to the variables of uart_handle_t . The variable contains the configuration information on a specified UART. |
| Return Value | <p>UART error code. The parameter can be one of the following values:</p> <ul style="list-style-type: none"> • HAL_UART_ERROR_NONE (no error) • HAL_UART_ERROR_PE (parity error) • HAL_UART_ERROR_FE (frame error) • HAL_UART_ERROR_OE (overflow error) • HAL_UART_ERROR_BI (line breaking error) • HAL_UART_ERROR_DMA (DMA transfer error) |

| | |
|---------|--|
| | <ul style="list-style-type: none"> • HAL_UART_ERROR_BUSY (busy) |
| Remarks | |

2.23.4.29 hal_uart_suspend_reg

Table 2-395 hal_uart_suspend_reg API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_uart_suspend_reg(uart_handle_t *p_uart) |
| Function Description | Suspend registers related to UART configuration before the device enters sleep mode. |
| Parameter | p_uart: pointer to the variables of uart_handle_t . The variable contains the configuration information on a specified UART. |
| Return Value | HAL status |

2.23.4.30 hal_uart_resume_reg

Table 2-396 hal_uart_resume_reg API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_uart_resume_reg(uart_handle_t *p_uart) |
| Function Description | Resume registers related to UART configuration during wakeup. |
| Parameter | p_uart: pointer to the variables of uart_handle_t . The variable contains the configuration information on a specified UART. |
| Return Value | HAL status |
| Remarks | |

2.24 HAL I2S Generic Driver

2.24.1 I2S Driver Functionalities

HAL Inter-IC Sound (I2S) driver features the following functionalities:

- I2S protocol introduced by Philips Semiconductor (now NXP Semiconductors)
- Independent TX and RX in full-duplex
- Master mode and slave mode
- Sound data resolution (bit): 12, 16, 20, 24, 32
- Three operating modes: polling, interrupt, and DMA
- Aborting data TX and RX/read and write in interrupt/DMA mode
- TX and RX complete interrupt callback function in master/slave mode
- Abort complete and I/O error interrupt callback functions
- To get the I2S configuration, operating state, and error code of the I2S driver

2.24.2 How to Use I2S Driver

Developers can:

1. Define the structure variable of `i2s_handle_t`, such as `i2s_handle_t i2s_handle` (`i2s_handle_t` structure is defined by HAL I2S driver. Developers shall define a variable for this type of handle structure before use.)
2. Initialize the I2S low-level resources by overwriting `hal_i2s_msp_init()`:
 - (1). Configure I2S GPIOs for functionality multiplexing and enable pull-up resistors.
 - (2). If I/O APIs in interrupt mode or DMA mode are required, you need to make NVIC configurations:
 - Configure the I2S interrupt priority by calling `hal_nvic_set_priority()`.
 - Enable NVIC interrupt for I2S by calling `hal_nvic_enable_irq()`.
 - (3). If I/O APIs are required to be operated in DMA mode, you also need to configure the necessary DMA channels:
 - Define `dma_handle_t` handle structure variables for transmission/reception, such as `dma_handle_t dma_tx` and `dma_handle_t dma_rx`.
 - Configure parameters of DMA handle (`dma_tx` and `dma_rx`), for example, specifying TX or RX channels.
 - Point `p_dmatx` and `p_dmarx` (variables of `i2s_handler` structure) to `dma_tx` and `dma_rx`, the initialized variables of DMA handle.
 - Configure the DMA interrupt priority, and enable NVIC interrupts for DMA.
3. Configure the data TX width, clock source, and audio frequency of the initialized I2S handle structure.
4. Configure I2S registers by calling `hal_i2s_init()`. During configuration, `hal_i2s_init()` automatically calls the overwritten `hal_i2s_msp_init()`, to initialize GPIOs and other low-level resources of I2S.
5. Three modes for SPI I/O operations (data read/write): polling, interrupt, and DMA.

2.24.2.1 I/O Read and Write in Polling Mode

1. Transmit a large volume of data in polling mode by running `hal_i2s_transmit()`.
2. Receive a large volume of data in polling mode by running `hal_i2s_receive()`.
3. Receive and transmit a large volume of data in polling mode by running `hal_i2s_transmit_receive()`.

2.24.2.2 I/O Read and Write in Interrupt Mode

1. Transmit a large volume of data in interrupt mode by running `hal_i2s_transmit_it()`. When a TX completes, `hal_i2s_tx_cplt_callback()` will be called. You can overwrite the callback for certain operations.
2. Receive a large volume of data in interrupt mode by running `hal_i2s_receive_it()`. When an RX completes, `hal_i2s_rx_cplt_callback()` will be called. You can overwrite the callback for certain operations.

3. Transmit and receive a large volume of data in interrupt mode by running `hal_i2s_transmit_receive_it()`. When RX and TX complete, `hal_i2s_tx_rx_cplt_callback()` will be called. You can overwrite the callback for certain operations.
4. If errors occur during data TX/RX, `hal_i2s_error_callback()` will be called. You can overwrite the callback for certain operations.
5. If you wish to abort data TX/RX, run `hal_i2s_abort()`.

2.24.2.3 I/O Read and Write in DMA Mode

1. Transmit a large volume of data in DMA mode by running `hal_i2s_transmit_dma()` as the master. When a TX completes, `hal_i2s_tx_cplt_callback()` will be called. You can overwrite the callback for certain operations.
2. Receive a large volume of data in DMA mode by running `hal_i2s_receive_dma()` as the master. When an RX completes, `hal_i2s_rx_cplt_callback()` will be called. You can overwrite the callback for certain operations.
3. Transmit and receive a large volume of data in DMA mode by running `hal_i2s_transmit_receive_dma()` as the mater. When an RX completes, `hal_i2s_tx_rx_cplt_callback()` will be called. You can overwrite the callback for certain operations.
4. If errors occur during data TX/RX, `hal_i2s_error_callback()` will be called. You can overwrite the callback for certain operations.

2.24.3 I2S Driver Structures

2.24.3.1 i2s_init_t

The initialization structure `i2s_init_t` of I2S driver is defined below:

Table 2-397 `i2s_init_t` structure

| Data Field | Field Description | Value |
|---------------------------------|-------------------|---|
| <code>uint32_t data_size</code> | Data TX width | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • <code>I2S_DATASIZE_12BIT</code> (12 bits) • <code>I2S_DATASIZE_16BIT</code> (16 bits) • <code>I2S_DATASIZE_20BIT</code> (20 bits) • <code>I2S_DATASIZE_24BIT</code> (24 bits) • <code>I2S_DATASIZE_32BIT</code> (32 bits) <p>Note:</p> <ul style="list-style-type: none"> • When <code>data_size = I2S_DATASIZE_12BIT</code> (12 bits), the transmitted data is 16-bit aligned and stored, with the higher 4-bit data ignored. The word sample size (WSS) of hardware is 16 SCLK cycles, with the higher 4-bit ignored. |

| Data Field | Field Description | Value |
|-----------------------|-------------------|--|
| | | <ul style="list-style-type: none"> When data_size = I2S_DATASIZE_20BIT (20 bits), the transmitted data is 32-bit aligned and stored, with the higher 12-bit data ignored. The WSS of hardware is 24 SCLK cycles, with the higher 4-bit ignored. When data_size = I2S_DATASIZE_24BIT (24 bits), the transmitted data is 32-bit aligned and stored, with the higher 8-bit data ignored. The WSS of hardware is 24 SCLK cycles. |
| uint32_t clock_source | Clock source | This parameter can be one of the following values: <ul style="list-style-type: none"> I2S_CLOCK_SRC_96M I2S_CLOCK_SRC_32M |
| uint32_t audio_freq | Audio frequency | audio_freq = fsck/(2 x wss), in which fsck means the serial clock frequency of I2S and can reach up to 3,027 kHz. WSS can be 16 bits, 24 bits, or 32 bits, depending on the bit width. When the bit width is configured to 16 bits, WSS is 16 bits, and audio_freq can be configured up to 96 kHz. |

2.24.3.2 i2s_handle_t

The i2s_handle_t structure of I2S driver is defined below:

Table 2-398 i2s_handle_t structure

| Data Field | Field Description | Value |
|-----------------------------|---|---|
| i2s_regs_t *p_instance | I2S instance | This parameter can be one of the following values: <ul style="list-style-type: none"> I2S_M I2S_S |
| i2s_init_t init | Initialization structure | See " Section 2.24.3.1 i2s_init_t ". |
| uint16_t *p_tx_buffer | Pointer to data TX buffer (managed by I2S driver and initialization by developers not required) | N/A |
| __IO uint32_t tx_xfer_size | Data TX size (managed by I2S driver and initialization by developers not required) | N/A |
| __IO uint32_t tx_xfer_count | Data TX count (managed by I2S driver and initialization by developers not required) | N/A |
| uint16_t *p_rx_buffer | Pointer to data RX buffer (managed by I2S driver and | N/A |

| Data Field | Field Description | Value |
|---|---|--|
| | initialization by developers not required) | |
| <code>__IO uint32_t rx_xfer_size</code> | Data RX size (managed by I2S driver and initialization by developers not required) | N/A |
| <code>__IO uint32_t rx_xfer_count</code> | Data RX count (managed by I2S driver and initialization by developers not required) | N/A |
| <code>void (*write_fifo)(struct_i2s_handle *p_i2s)</code> | Pointer to the write FIFO functions of I2S TX (managed by I2S driver and initialization by developers not required) | N/A |
| <code>void (*read_fifo)(struct_i2s_handle *p_i2s)</code> | Pointer to the read FIFO functions of I2S RX (managed by I2S driver and initialization by developers not required) | N/A |
| <code>dma_handle_t *p_dmatx</code> | DMA handle pointer to I2S TX channel | Structure of DMA handle <code>dma_handle_t</code> for TX channels |
| <code>dma_handle_t *p_dmarx</code> | DMA handle pointer to I2S RX channel | Structure of DMA handle <code>dma_handle_t</code> for RX channels |
| <code>__IO hal_lock_t lock</code> | I2S lock (managed by I2S driver and initialization by developers not required) | N/A |
| <code>__IO hal_i2s_state_t state</code> | I2S operating state (initialization by developers not required) | This parameter can be one of the following values: <ul style="list-style-type: none"> • HAL_I2S_STATE_RESET (not initialized) • HAL_I2S_STATE_READY (initialized and ready for use) • HAL_I2S_STATE_BUSY (busy) • HAL_I2S_STATE_BUSY_TX (TX ongoing) • HAL_I2S_STATE_BUSY_RX (RX ongoing) • HAL_I2S_STATE_BUSY_TX_RX (TX and RX ongoing) • HAL_I2S_STATE_ABORT (aborted) • HAL_I2S_STATE_ERROR (error) |
| <code>__IO uint32_t error_code</code> | I2S error code (initialization by developers not required) | This parameter can be one of the following values: <ul style="list-style-type: none"> • HAL_I2S_ERROR_NONE (no error) |

| Data Field | Field Description | Value |
|------------|-------------------|---|
| | | <ul style="list-style-type: none"> • HAL_I2S_ERROR_TIMEOUT (timeout) • HAL_I2S_ERROR_TRANSFER (transfer error) • HAL_I2S_ERROR_DMA (DMA transfer error) • HAL_I2S_ERROR_INVALID_PARAM (invalid parameter) |

2.24.4 I2S Driver APIs

The I2S driver APIs are listed in the table below:

Table 2-399 I2S driver APIs

| API Type | API Name | Description |
|---------------------------------|--------------------------------|--|
| Initialization | hal_i2s_init() | Initialize I2S peripheral, and configure parameters. |
| | hal_i2s_deinit() | Deinitialize I2S peripheral. |
| | hal_i2s_msp_init() | Initialize GPIO pin multiplexing, NVIC interrupts, and DMA channels of I2S peripheral. |
| | hal_i2s_msp_deinit() | Deinitialize GPIO pin multiplexing, NVIC interrupts, and DMA channels of I2S peripheral. |
| I/O operation | hal_i2s_transmit() | Transmit data in polling mode. |
| | hal_i2s_receive() | Receive data in polling mode. |
| | hal_i2s_transmit_receive() | Transmit and receive data in polling mode. |
| | hal_i2s_transmit_it() | Transmit data in interrupt mode. |
| | hal_i2s_receive_it() | Receive data in interrupt mode. |
| | hal_i2s_transmit_receive_it() | Transmit and receive data in interrupt mode. |
| | hal_i2s_transmit_dma() | Transmit data in DMA mode. |
| | hal_i2s_receive_dma() | Receive data in DMA mode. |
| | hal_i2s_transmit_receive_dma() | Transmit and receive data in DMA mode. |
| | hal_i2s_abort() | Abort data transfer in interrupt/DMA mode. |
| Interrupt handling and callback | hal_i2s_irq_handler() | Interrupt handler |
| | hal_i2s_tx_cplt_callback() | TX complete interrupt callback |
| | hal_i2s_rx_cplt_callback() | RX complete interrupt callback |
| | hal_i2s_error_callback() | Error interrupt callback |
| State and error | hal_i2s_get_state() | Get the driver operating state. |
| | hal_i2s_get_error() | Get error code. |
| Clock control | hal_i2s_start_clock() | Start clock output in master mode. |
| | hal_i2s_stop_clock() | Stop clock output in master mode. |
| Functions about FIFO threshold | hal_i2s_set_tx_fifo_threshold | Set a TX FIFO threshold. |

| API Type | API Name | Description |
|----------|-------------------------------|---|
| | hal_i2s_set_rx_fifo_threshold | Set an RX FIFO threshold. |
| | hal_i2s_get_tx_fifo_threshold | Get a TX FIFO threshold. |
| | hal_i2s_get_rx_fifo_threshold | Get an RX FIFO threshold. |
| Sleep | hal_i2s_suspend_reg() | Suspend registers related to I2S configuration before the device enters sleep mode. |
| | hal_i2s_resume_reg() | Resume registers related to I2S configuration during wakeup. |

The sections below elaborate on these APIs.

2.24.4.1 hal_i2s_init

Table 2-400 hal_i2s_init API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_i2s_init(i2s_handle_t *p_i2s) |
| Function Description | Initialize I2S and relevant handles based on specified parameters in i2s_init_t . |
| Parameter | p_i2s: pointer to the variables of i2s_handle_t . The variable contains the configuration information on a specified I2S. |
| Return Value | HAL status |
| Remarks | |

2.24.4.2 hal_i2s_deinit

Table 2-401 hal_i2s_deinit API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_i2s_deinit(i2s_handle_t *p_i2s) |
| Function Description | Deinitialize I2S. |
| Parameter | p_i2s: pointer to the variables of i2s_handle_t . The variable contains the configuration information on a specified I2S. |
| Return Value | HAL status |
| Remarks | |

2.24.4.3 hal_i2s_msp_init

Table 2-402 hal_i2s_msp_init API

| | |
|-----------------------------|---|
| Function Prototype | void hal_i2s_msp_init(i2s_handle_t *p_i2s) |
| Function Description | Initialize the GPIO pin multiplexing, NVIC interrupts, and DMA channels of I2S. |
| Parameter | p_i2s: pointer to the variables of i2s_handle_t . The variable contains the configuration information on a specified I2S. |

| | |
|---------------------|---|
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite the API to initialize GPIO pin multiplexing, NVIC interrupts, and DMA channels. |

2.24.4.4 hal_i2s_msp_deinit

Table 2-403 hal_i2s_msp_deinit API

| | |
|-----------------------------|---|
| Function Prototype | void hal_i2s_msp_deinit(i2s_handle_t *p_i2s) |
| Function Description | Deinitialize the GPIO pin multiplexing, NVIC interrupts, and DMA channels I2S. |
| Parameter | p_i2s: pointer to the variables of i2s_handle_t . The variable contains the configuration information on a specified I2S. |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite the API to deinitialize GPIO pin multiplexing, NVIC interrupts, and DMA channels. |

2.24.4.5 hal_i2s_transmit

Table 2-404 hal_i2s_transmit API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_i2s_transmit(i2s_handle_t *p_i2s, uint16_t *p_data, uint32_t length, uint32_t timeout) |
| Function Description | Transmit a large volume of data in polling mode. |
| Parameter | p_i2s: pointer to the variables of i2s_handle_t . The variable contains the configuration information on a specified I2S. p_data: pointer to data buffer length: length of single-track data to be transmitted (unit: 2 bytes) timeout: timeout period (ms) |
| Return Value | HAL status |
| Remarks | When it returns HAL_ERROR, you can call hal_i2s_get_error() to retrieve the error code. |

2.24.4.6 hal_i2s_receive

Table 2-405 hal_i2s_receive API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_i2s_receive(i2s_handle_t *p_i2s, uint16_t *p_data, uint32_t length, uint32_t timeout) |
| Function Description | Receive a large amount of data in polling mode. |
| Parameter | p_i2s: pointer to the variables of i2s_handle_t . The variable contains the configuration information on a specified I2S. p_data: pointer to data buffer length: length of data to be received; single-track data (unit: 2 bytes) |

| | |
|---------------------|---|
| | timeout: timeout period (ms) |
| Return Value | HAL status |
| Remarks | When it returns HAL_ERROR, you can call hal_i2s_get_error() to retrieve the error code. |

2.24.4.7 hal_i2s_transmit_receive

Table 2-406 hal_i2s_transmit_receive API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_i2s_transmit_receive(i2s_handle_t *p_i2s, uint16_t *p_tx_data, uint16_t *p_rx_data, uint32_t length, uint32_t timeout) |
| Function Description | Transmit and receive a large amount of data in polling mode. |
| Parameter | <p>p_i2s: pointer to the variables of i2s_handle_t. The variable contains the configuration information on a specified I2S.</p> <p>p_tx_data: pointer to data TX buffer</p> <p>p_rx_data: pointer to data RX buffer</p> <p>length: length of transmitted and received data (unit: byte)</p> <p>timeout: timeout period (ms)</p> |
| Return Value | HAL status |
| Remarks | When it returns HAL_ERROR, you can call hal_i2s_get_error() to retrieve the error code. |

2.24.4.8 hal_i2s_transmit_it

Table 2-407 hal_i2s_transmit_it API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_i2s_transmit_it(i2s_handle_t *p_i2s, uint16_t *p_data, uint32_t length) |
| Function Description | Transmit a large volume of data in interrupt mode. |
| Parameter | <p>p_i2s: pointer to the variables of i2s_handle_t. The variable contains the configuration information on a specified I2S.</p> <p>p_data: pointer to data buffer</p> <p>length: length of single-track data to be transmitted (unit: 2 bytes)</p> |
| Return Value | HAL status |
| Remarks | <ul style="list-style-type: none"> When TX completes, the callback function hal_i2s_tx_cplt_callback() will be called. When an error occurs during TX, the callback function hal_i2s_error_callback() will be called. The related error code can be retrieved by calling hal_i2s_get_error() in the callback function. Before calling hal_i2s_tx_cplt_callback(), do not release the memory of the data buffer pointed by data. |

2.24.4.9 hal_i2s_receive_it

Table 2-408 hal_i2s_receive_it API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_i2s_receive_it(i2s_handle_t *p_i2s, uint16_t *p_data, uint32_t length) |
| Function Description | Receive a large amount of data in interrupt mode. |
| Parameter | <p>p_i2s: pointer to the variables of i2s_handle_t. The variable contains the configuration information on a specified I2S.</p> <p>p_data: pointer to data buffer</p> <p>length: length of data to be received; single-track data (unit: 2 bytes)</p> |
| Return Value | HAL status |
| Remarks | <ul style="list-style-type: none"> When RX completes, the callback function hal_i2s_rx_cplt_callback() will be called. When an error occurs during RX, the callback function hal_i2s_error_callback() will be called. The related error code can be retrieved by calling hal_i2s_get_error() in the callback function. Before calling hal_i2s_rx_cplt_callback(), do not release the memory of the data buffer pointed by data. |

2.24.4.10 hal_i2s_transmit_receive_it

Table 2-409 hal_i2s_transmit_receive_it API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_i2s_transmit_receive_it(i2s_handle_t *p_i2s, uint16_t *p_tx_data, uint16_t *p_rx_data, uint32_t length) |
| Function Description | Transmit and receive a large amount of data in interrupt mode. |
| Parameter | <p>p_i2s: pointer to the variables of i2s_handle_t. The variable contains the configuration information on a specified I2S.</p> <p>p_tx_data: pointer to data TX buffer</p> <p>p_rx_data: pointer to data RX buffer</p> <p>length: length of transmitted and received data (unit: byte)</p> |
| Return Value | HAL status |
| Remarks | <ul style="list-style-type: none"> When RX completes, the callback function hal_i2s_tx_rx_cplt_callback () will be called. When an error occurs during RX/TX, the callback function hal_i2s_error_callback() will be called. The related error code can be retrieved by calling hal_i2s_get_error() in the callback function. Before calling hal_i2s_rx_cplt_callback(), do not release the memory of the data buffer pointed by data. |

2.24.4.11 hal_i2s_abort

Table 2-410 hal_i2s_abort API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_i2s_abort(i2s_handle_t *p_i2s) |
| Function Description | Abort I2S data transfer in interrupt/DMA mode. |
| Parameter | p_i2s: pointer to the variables of i2s_handle_t . The variable contains the configuration information on a specified I2S. |

| | |
|---------------------|---|
| Return Value | HAL status |
| Remarks | This is a polling function. It exits from the function when an abort completes. |

2.24.4.12 hal_i2s_transmit_dma

Table 2-411 hal_i2s_transmit_dma API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_i2s_transmit_dma(i2s_handle_t *p_i2s, uint16_t *p_data, uint32_t length) |
| Function Description | Transmit a large amount of data in DMA mode. |
| Parameter | p_i2s: pointer to the variables of i2s_handle_t . The variable contains the configuration information on a specified I2S. p_data: pointer to data buffer length: length of data to be transmitted |
| Return Value | HAL status |
| Remarks | <ul style="list-style-type: none"> When TX completes, the callback function hal_i2s_tx_cplt_callback() will be called. When an error occurs during TX, the callback function hal_i2s_error_callback() will be called. The related error code can be retrieved by calling hal_i2s_get_error() in the callback function. Before calling hal_i2s_tx_cplt_callback(), do not release the memory of the data buffer pointed by data. |

2.24.4.13 hal_i2s_receive_dma()

Table 2-412 hal_i2s_receive_dma API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_i2s_receive_dma(i2s_handle_t *p_i2s, uint16_t *p_data, uint32_t length) |
| Function Description | Receive a large amount of data in DMA mode. |
| Parameter | p_i2s: pointer to the variables of i2s_handle_t . The variable contains the configuration information on a specified I2S. p_data: pointer to data buffer length: length of data to be received |
| Return Value | HAL status |
| Remarks | <ul style="list-style-type: none"> When RX completes, the callback function hal_i2s_rx_cplt_callback() will be called. When an error occurs during RX, the callback function hal_i2s_error_callback() will be called. The related error code can be retrieved by calling hal_i2s_get_error() in the callback function. Before calling hal_i2s_rx_cplt_callback(), do not release the memory of the data buffer pointed by data. |

2.24.4.14 hal_i2s_transmit_receive_dma

Table 2-413 hal_i2s_transmit_receive_dma API

| | |
|---------------------------|---|
| Function Prototype | hal_status_t hal_i2s_transmit_receive_dma(i2s_handle_t *p_i2s, uint16_t *p_tx_data, uint16_t *p_rx_data, uint32_t length) |
|---------------------------|---|

| | |
|-----------------------------|--|
| Function Description | Transmit and receive a large volume of data in DMA mode. |
| Parameter | <p>p_i2s: pointer to the variables of i2s_handle_t. The variable contains the configuration information on a specified I2S.</p> <p>p_tx_data: pointer to data TX buffer</p> <p>p_rx_data: pointer to data RX buffer</p> <p>length: length of transmitted and received data (unit: byte)</p> |
| Return Value | HAL status |
| Remarks | <ul style="list-style-type: none"> When RX completes, the callback function hal_i2s_tx_rx_cplt_callback() will be called. When an error occurs during RX/TX, the callback function hal_i2s_error_callback() will be called. The related error code can be retrieved by calling hal_i2s_get_error() in the callback function. Before calling hal_i2s_rx_cplt_callback(), do not release the memory of the data buffer pointed by data. |

2.24.4.15 hal_i2s_irq_handler

Table 2-414 hal_i2s_irq_handler API

| | |
|-----------------------------|---|
| Function Prototype | void hal_i2s_irq_handler(i2s_handle_t *p_i2s) |
| Function Description | Handle I2S interrupt requests. |
| Parameter | p_i2s: pointer to the variables of i2s_handle_t . The variable contains the configuration information on a specified I2S. |
| Return Value | None |
| Remarks | |

2.24.4.16 hal_i2s_tx_cplt_callback

Table 2-415 hal_i2s_tx_cplt_callback API

| | |
|-----------------------------|---|
| Function Prototype | void hal_i2s_tx_cplt_callback(i2s_handle_t *p_i2s) |
| Function Description | TX complete callback function |
| Parameter | p_i2s: pointer to the variables of i2s_handle_t . The variable contains the configuration information on a specified I2S. |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite this function before using it. |

2.24.4.17 hal_i2s_tx_rx_cplt_callback

Table 2-416 hal_i2s_tx_rx_cplt_callback API

| | |
|---------------------------|---|
| Function Prototype | void hal_i2s_tx_rx_cplt_callback(i2s_handle_t *p_i2s) |
|---------------------------|---|

| | |
|-----------------------------|---|
| Function Description | TX and RX complete callback function |
| Parameter | p_i2s: pointer to the variables of i2s_handle_t . The variable contains the configuration information on a specified I2S. |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite this function before using it. |

2.24.4.18 hal_i2s_rx_cplt_callback

Table 2-417 hal_i2s_rx_cplt_callback API

| | |
|-----------------------------|---|
| Function Prototype | void hal_i2s_rx_cplt_callback(i2s_handle_t *p_i2s) |
| Function Description | RX complete callback function |
| Parameter | p_i2s: pointer to the variables of i2s_handle_t . The variable contains the configuration information on a specified I2S. |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite this function before using it. |

2.24.4.19 hal_i2s_error_callback

Table 2-418 hal_i2s_error_callback API

| | |
|-----------------------------|---|
| Function Prototype | void hal_i2s_error_callback(i2s_handle_t *p_i2s) |
| Function Description | I2S error callback function |
| Parameter | p_i2s: pointer to the variables of i2s_handle_t . The variable contains the configuration information on a specified I2S. |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite this function before using it. |

2.24.4.20 hal_i2s_get_state

Table 2-419 hal_i2s_get_state API

| | |
|-----------------------------|---|
| Function Prototype | hal_i2s_state_t hal_i2s_get_state(i2s_handle_t *p_i2s) |
| Function Description | Get the operating state of I2S. |
| Parameter | p_i2s: pointer to the variables of i2s_handle_t . The variable contains the configuration information on a specified I2S. |
| Return Value | I2S operating state, which can be one of the following values: |

| | |
|---------|---|
| | <ul style="list-style-type: none"> • HAL_I2S_STATE_RESET (not initialized) • HAL_I2S_STATE_READY (initialized and ready for use) • HAL_I2S_STATE_BUSY (busy) • HAL_I2S_STATE_BUSY_TX (TX ongoing) • HAL_I2S_STATE_BUSY_RX (RX ongoing) • HAL_I2S_STATE_BUSY_TX_RX (TX and RX ongoing) • HAL_I2S_STATE_ABORT (aborted) • HAL_I2S_STATE_ERROR (error) |
| Remarks | |

2.24.4.21 hal_i2s_get_error

Table 2-420 hal_i2s_get_error API

| | |
|-----------------------------|--|
| Function Prototype | uint32_t hal_i2s_get_error(i2s_handle_t *p_i2s) |
| Function Description | Return the I2S error code. |
| Parameter | p_i2s: pointer to the variables of i2s_handle_t . The variable contains the configuration information on a specified I2S. |
| Return Value | <p>I2S error code can be one of the following values:</p> <ul style="list-style-type: none"> • HAL_I2S_ERROR_NONE (no error) • HAL_I2S_ERROR_TIMEOUT (timeout) • HAL_I2S_ERROR_TRANSFER (transfer error) • HAL_I2S_ERROR_DMA (DMA transfer error) • HAL_I2S_ERROR_INVALID_PARAM (invalid parameter) |
| Remarks | |

2.24.4.22 hal_i2s_start_clock

Table 2-421 hal_i2s_start_clock API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_i2s_start_clock(i2s_handle_t *p_i2s) |
| Function Description | Start clock output (as the master). |
| Parameter | p_i2s: pointer to the variables of i2s_handle_t . The variable contains the configuration information on a specified I2S. |
| Return Value | HAL status |
| Remarks | |

2.24.4.23 hal_i2s_stop_clock

Table 2-422 hal_i2s_stop_clock API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_i2s_stop_clock(i2s_handle_t *p_i2s) |
| Function Description | Stop clock output (as the master). |
| Parameter | p_i2s: pointer to the variables of i2s_handle_t . The variable contains the configuration information on a specified I2S. |
| Return Value | HAL status |
| Remarks | |

2.24.4.24 hal_i2s_set_tx_fifo_threshold

Table 2-423 hal_i2s_set_tx_fifo_threshold API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_i2s_set_tx_fifo_threshold(i2s_handle_t *p_i2s, uint32_t threshold) |
| Function Description | Set a TX FIFO threshold. |
| Parameter | p_i2s: pointer to the variables of i2s_handle_t . The variable contains the configuration information on a specified I2S. threshold: a threshold to be set |
| Return Value | HAL status |
| Remarks | |

2.24.4.25 hal_i2s_set_rx_fifo_threshold

Table 2-424 hal_i2s_set_rx_fifo_threshold API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_i2s_set_rx_fifo_threshold(i2s_handle_t *p_i2s, uint32_t threshold) |
| Function Description | Set an RX FIFO threshold. |
| Parameter | p_i2s: pointer to the variables of i2s_handle_t . The variable contains the configuration information on a specified I2S. threshold: a threshold to be set |
| Return Value | HAL status |
| Remarks | |

2.24.4.26 hal_i2s_get_tx_fifo_threshold

Table 2-425 hal_i2s_get_tx_fifo_threshold API

| | |
|-----------------------------|---|
| Function Prototype | uint32_t hal_i2s_get_tx_fifo_threshold(i2s_handle_t *p_i2s) |
| Function Description | Get a TX FIFO threshold. |
| Parameter | p_i2s: pointer to the variables of i2s_handle_t . The variable contains the configuration information on a specified I2S. |

| | |
|---------------------|-------------------|
| Return Value | TX FIFO threshold |
| Remarks | |

2.24.4.27 hal_i2s_get_rx_fifo_threshold

Table 2-426 hal_i2s_get_rx_fifo_threshold API

| | |
|-----------------------------|---|
| Function Prototype | uint32_t hal_i2s_get_tx_fifo_threshold(i2s_handle_t *p_i2s) |
| Function Description | Get an RX FIFO threshold. |
| Parameter | p_i2s: pointer to the variables of i2s_handle_t . The variable contains the configuration information on a specified I2S. |
| Return Value | RX FIFO threshold |
| Remarks | |

2.24.4.28 hal_i2s_suspend_reg

Table 2-427 hal_i2s_suspend_reg API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_i2s_suspend_reg(i2s_handle_t *p_i2s) |
| Function Description | Suspend registers related to I2S configuration before the device enters sleep mode. |
| Parameter | p_i2s: pointer to the variables of i2s_handle_t . The variable contains the configuration information on a specified I2S. |
| Return Value | HAL status |
| Remarks | |

2.24.4.29 hal_i2s_resume_reg

Table 2-428 hal_i2s_resume_reg API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_i2s_resume_reg(i2s_handle_t *p_i2s) |
| Function Description | Resume registers related to I2S configuration during wakeup. |
| Parameter | p_i2s: pointer to the variables of i2s_handle_t . The variable contains the configuration information on a specified I2S. |
| Return Value | HAL status |
| Remarks | |

2.25 HAL RNG Generic Driver

2.25.1 RNG Driver Functionalities

The HAL Random Number Generator (RNG) driver features the following functionalities:

- Generate true random numbers and pseudo-random numbers.
- The generated random numbers can pass NIST SP 800-22, a standard test suite.
- Support multiple post-processing methods, including skipping, bit counting, and Von Neumann architecture.
- Operate in interrupt/polling mode.

2.25.2 How to Use RNG Driver

Developers can use RNG driver in the following scenarios:

1. Overwrite `hal_rng_msp_init()`, and call `hal_nvic_set_priority()` and `hal_nvic_enable_irq()` in the overwritten API to enable NVIC interrupt for RNG.
2. Declare a structure variable of `rng_handle_t`, for example: `rng_handle_t p_rng`, and set the `p_instance` member as RNG instances.
3. Configure the initial count and the reset mode for the initialization structure of `p_rng`.
4. Initialize RNG peripheral by calling `hal_rng_init()`.
5. Generate random numbers in polling mode by calling `hal_rng_generate_random_number()`, or in interrupt mode by calling `hal_rng_generate_random_number_it()`. If you choose `RNG_SEED_USER`, 59-bit or 128-bit random number seeds are required.
6. After random numbers are generated in interrupt mode, `hal_rng_ready_data_callback()` will be called. You can overwrite the API if necessary.

2.25.3 RNG Driver Structures

2.25.3.1 rng_init_t

The initialization structure `rng_init_t` of RNG driver is defined below:

Table 2-429 `rng_init_t` structure

| Data Field | Field Description | Value |
|---------------------------------|--|--|
| <code>uint32_t seed_mode</code> | Defining linear-feedback shift register (LFSR) seeds | This parameter can be one of the following values: <ul style="list-style-type: none"> • <code>RNG_SEED_FRO_S0</code> (LFSR seed generated by switching oscillator s0) • <code>RNG_SEED_USER</code> (LFSR seed configured by users) |
| <code>uint32_t lfsr_mode</code> | LFSR configuration mode | This parameter can be one of the following values: <ul style="list-style-type: none"> • <code>RNG_LFSR_MODE_59BIT</code> (59-bit LFSR) • <code>RNG_LFSR_MODE_128BIT</code> (128-bit LFSR) |
| <code>uint32_t out_mode</code> | Random number output mode | This parameter can be one of the following values: <ul style="list-style-type: none"> • <code>RNG_OUTPUT_FRO_S0</code> (numbers directly from RNG) |

| Data Field | Field Description | Value |
|--------------------|----------------------|--|
| | | <ul style="list-style-type: none"> RNG_OUTPUT_CYCLIC_PARITY (cyclic sampling from LFSR and RNG, and odd-even parity check) RNG_OUTPUT_CYCLIC (cyclic sampling from LFSR and RNG) RNG_OUTPUT_LFSR_RNG (LFSR \oplus RNG) RNG_OUTPUT_LFSR (direct output from LFSR) <p>Note: When seed_mode is set as RNG_SEED_USER, out_mode cannot be set as RNG_OUTPUT_FRO_S0.</p> |
| uint32_t post_mode | Post-processing mode | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> RNG_POST_PRO_NOT (no post-processing) RNG_POST_PRO_SKIPPING (skipping) RNG_OUTPUT_CYCLIC (bit counting) RNG_OUTPUT_LFSR_RNG (Von Neumann architecture) |

2.25.3.2 rng_handle_t

The handle structure rng_handle_t of RNG driver is defined below:

Table 2-430 rng_handle_t structure

| Data Field | Field Description | Value |
|------------------------------|--|---|
| rng_regs_t *p_instance | RNG peripheral instance | Random numbers from RNG |
| rng_init_t init | Initialization structure | See " Section 2.25.3.1 rng_init_t ". |
| __IO hal_lock_t lock | RNG lock (initialization by developers not required) | N/A |
| __IO hal_rng_state_t g_state | RNG operating state (initialization by developers not required) | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> HAL_RNG_STATE_RESET (not initialized) HAL_RNG_STATE_READY (initialized and ready for use) HAL_RNG_STATE_BUSY (busy) HAL_RNG_STATE_TIMEOUT (timeout) HAL_RNG_STATE_ERROR (error) |
| uint32_t random_number | Last generated random number | Range: 0x00000000 to 0xFFFFFFFF |
| uint32_t retention[1] | RNG register information (managed by RNG driver and initialization by developers not required) | N/A |

2.25.4 RNG Driver APIs

The RNG driver APIs are listed in the table below:

Table 2-431 RNG driver APIs

| API Type | API Name | Description |
|---------------------------------|-----------------------------------|---|
| Initialization | hal_rng_init() | Initialize RNG peripheral, and configure the initial count and other parameters. |
| | hal_rng_deinit() | Deinitialize RNG peripheral. |
| | hal_rng_msp_init() | Initialize NVIC interrupts of RNG. |
| | hal_rng_msp_deinit() | Deinitialize NVIC interrupts of RNG. |
| I/O operation | hal_rng_generate_random_number | Generate random numbers in polling mode. |
| | hal_rng_generate_random_number_it | Generate random numbers in interrupt mode. |
| | hal_rng_read_last_random_number | Get the last generated random number. |
| Interrupt handling and callback | hal_rng_irq_handler | Interrupt handler |
| | hal_rng_ready_data_callback | Counting complete callback function |
| Sleep | hal_rng_suspend_reg() | Suspend registers related to RNG configuration before the device enters sleep mode. |
| | hal_rng_resume_reg() | Resume registers related to RNG configuration during wakeup. |

The sections below elaborate on these APIs.

2.25.4.1 hal_rng_init

Table 2-432 hal_rng_init API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_rng_init(rng_handle_t *p_rng) |
| Function Description | Initialize RNG according to the specified parameters in RNG handle. |
| Parameter | p_rng: pointer to variables of rng_handle_t . The variable contains the configuration information on a specified RNG. |
| Return Value | HAL status |
| Remarks | |

2.25.4.2 hal_rng_deinit

Table 2-433 hal_rng_deinit API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_rng_deinit(rng_handle_t *p_rng) |
| Function Description | Deinitialize RNG peripheral registers and set the register values as default. |

| | |
|---------------------|---|
| Parameter | p_rng: pointer to variables of rng_handle_t . The variable contains the configuration information on a specified RNG. |
| Return Value | HAL status |
| Remarks | |

2.25.4.3 hal_rng_msp_init

Table 2-434 hal_rng_msp_init API

| | |
|-----------------------------|---|
| Function Prototype | void hal_rng_msp_init(rng_handle_t *p_rng) |
| Function Description | Initialize NVIC interrupts of RNG. |
| Parameter | p_rng: pointer to variables of rng_handle_t . The variable contains the configuration information on a specified RNG. |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite the API to initialize the RNG interrupts. |

2.25.4.4 hal_rng_msp_deinit

Table 2-435 hal_rng_msp_deinit API

| | |
|-----------------------------|---|
| Function Prototype | void hal_rng_msp_deinit(rng_handle_t *p_rng) |
| Function Description | Deinitialize NVIC interrupts of RNG. |
| Parameter | p_rng: pointer to variables of rng_handle_t . The variable contains the configuration information on a specified RNG. |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite the API to disable the NVIC interrupts. |

2.25.4.5 hal_rng_generate_random_number

Table 2-436 hal_rng_generate_random_number API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_rng_generate_random_number(rng_handle_t *p_rng, uint16_t *p_seed, uint32_t *p_random32bit) |
| Function Description | Generate random numbers in polling mode. |
| Parameter | p_rng: pointer to variables of rng_handle_t . The variable contains the configuration information on a specified RNG. p_seed: pointer to the random number seed configured by users The parameter is only valid when seed_mode in rng_init_t is configured as RNG_SEED_USER. |

| | |
|---------------------|---|
| | p_random32bit: pointer to the last random number generated by the RNG |
| Return Value | HAL status |
| Remarks | |

2.25.4.6 hal_rng_generate_random_number_it

Table 2-437 hal_rng_generate_random_number_it API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_rng_generate_random_number_it(rng_handle_t *p_rng, uint16_t *p_seed, uint32_t *p_random32bit) |
| Function Description | Generate random numbers in interrupt mode. |
| Parameter | p_rng: pointer to variables of rng_handle_t . The variable contains the configuration information on a specified RNG. p_seed: pointer to the random number seed configured by users The parameter is only valid when seed_mode in rng_init_t is configured as RNG_SEED_USER. p_random32bit: pointer to the last random number generated by the RNG |
| Return Value | HAL status |
| Remarks | |

2.25.4.7 hal_rng_read_last_random_number

Table 2-438 hal_rng_read_last_random_number API

| | |
|-----------------------------|---|
| Function Prototype | uint32_t hal_rng_read_last_random_number(rng_handle_t *p_rng) |
| Function Description | Get the last generated random number. |
| Parameter | p_rng: pointer to variables of rng_handle_t . The variable contains the configuration information on a specified RNG. |
| Return Value | The last generated random number, range: 0x00000000 to 0xFFFFFFFF |
| Remarks | |

2.25.4.8 hal_rng_irq_handler

Table 2-439 hal_rng_irq_handler API

| | |
|-----------------------------|---|
| Function Prototype | void hal_rng_irq_handler(rng_handle_t *p_rng) |
| Function Description | Handle RNG interrupt requests. |
| Parameter | p_rng: pointer to variables of rng_handle_t . The variable contains the configuration information on a specified RNG. |
| Return Value | None |

| | |
|---------|--|
| Remarks | |
|---------|--|

2.25.4.9 hal_rng_ready_data_callback

Table 2-440 hal_rng_ready_data_callback API

| | |
|-----------------------------|--|
| Function Prototype | void hal_rng_ready_data_callback(rng_handle_t *p_rng, uint32_t random32bit) |
| Function Description | RNG generation complete interrupt callback function |
| Parameter | p_rng: pointer to variables of rng_handle_t . The variable contains the configuration information on a specified RNG. random32bit: random number generated in RNG |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite this function before using it. |

2.25.4.10 hal_rng_suspend_reg

Table 2-441 hal_rng_suspend_reg API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_rng_suspend_reg(rng_handle_t *p_rng); |
| Function Description | Suspend registers related to RNG configuration before the device enters sleep mode. |
| Parameter | p_rng: pointer to variables of rng_handle_t . The variable contains the configuration information on a specified RNG. |
| Return Value | HAL status |
| Remarks | |

2.25.4.11 hal_rng_resume_reg

Table 2-442 hal_rng_resume_reg API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_rng_resume_reg(rng_handle_t *p_rng) |
| Function Description | Resume registers related to RNG configuration during wakeup. |
| Parameter | p_rng: pointer to variables of rng_handle_t . The variable contains the configuration information on a specified RNG. |
| Return Value | HAL status |
| Remarks | |

2.26 HAL AON WDT Generic Driver

2.26.1 AON WDT Driver Functionalities

The HAL Always-on Watchdog Timer (AON WDT) driver features the following functionalities:

- Enable and disable the reset mode. When the reset mode is enabled, AON WDT triggers an interrupt when counting down to `alarm_counter`, and resets the system when counting down to 0.
- Reload the initial value of counting, which means to feed the watchdog.
- Interrupt callback function

2.26.2 How to Use AON WDT Driver

Developers can use AON WDT driver in the following scenarios:

1. Declare a structure variable of `aon_wdt_handle_t`, for example: `aon_wdt_handle_t hwdt`.
2. Configure the initial count and the reset mode for the initialization structure of `p_aon_wdt` handle.
3. Initialize AON WDT peripheral by calling `hal_aon_wdt_init()`.
4. Before the AON WDT counts down to 0, developers shall reload the initial count by calling `hal_aon_wdt_refresh()`, or the system will be reset by AON WDT automatically.
5. When AON WDT counts down to `alarm_counter`, the interrupt callback `hal_wdt_period_elapsed_callback()` will be called. Developers can overwrite the API if necessary.

2.26.3 AON WDT Driver Structures

2.26.3.1 aon_wdt_init_t

The initialization structure `aon_wdt_init_t` of AON WDT driver is defined below:

Table 2-443 aon_wdt_init_t structure

| Data Field | Field Description | Value |
|-------------------------------------|--------------------------------------|----------------------------|
| <code>uint32_t counter</code> | Initial count | 0x0000_0000 to 0xFFFF_FFFF |
| <code>uint32_t alarm_counter</code> | Initial value of counter reset alarm | 0 – 20 |

2.26.3.2 aon_wdt_handle_t

The structure `aon_wdt_handle_t` of AON WDT driver is defined below:

Table 2-444 aon_aon_wdt_handle_t structure

| Data Field | Field Description | Value |
|-----------------------------------|--|--|
| <code>aon_wdt_init_t init</code> | Initialization structure | See " Section 2.26.3.1 aon_wdt_init_t ". |
| <code>__IO hal_lock_t lock</code> | AON WDT lock (initialization by developers not required) | N/A |

2.26.4 AON WDT Driver APIs

The AON WDT driver APIs are listed in the table below:

Table 2-445 AON WDT driver APIs

| API Type | API Name | Description |
|---------------------------------|----------------------------|--|
| Initialization | hal_aon_wdt_init() | Initialize AON WDT peripheral, and configure the initial count and other parameters. |
| | hal_aon_wdt_deinit() | Deinitialize AON WDT peripheral. |
| I/O operation | hal_aon_wdt_refresh | Reload initial count. |
| Interrupt handling and callback | hal_aon_wdt_irq_handler | Interrupt handler |
| | hal_aon_wdt_alarm_callback | Callback function for reset alarm |

The sections below elaborate on these APIs.

2.26.4.1 hal_aon_wdt_init

Table 2-446 hal_aon_wdt_init API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_aon_wdt_init(aon_wdt_handle_t *p_aon_wdt) |
| Function Description | Initialize AON WDT according to specified parameters of AON WDT handle. |
| Parameter | p_aon_wdt: pointer to the variables of aon_wdt_handle_t . The variable contains the configuration information on a specified AON WDT. |
| Return Value | HAL status |
| Remarks | |

2.26.4.2 hal_aon_wdt_deinit

Table 2-447 hal_aon_wdt_deinit API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_aon_wdt_deinit(aon_wdt_handle_t *p_aon_wdt) |
| Function Description | Deinitialize the AON WDT peripheral registers and set the register values to default. |
| Parameter | p_aon_wdt: pointer to the variables of aon_wdt_handle_t . The variable contains the configuration information on a specified AON WDT. |
| Return Value | HAL status |
| Remarks | |

2.26.4.3 hal_aon_wdt_refresh

Table 2-448 hal_aon_wdt_refresh API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_aon_wdt_refresh (aon_wdt_handle_t *p_aon_wdt) |
| Function Description | Refresh AON WDT counts. |

| | |
|---------------------|---|
| Parameter | p_aon_wdt: pointer to the variables of aon_wdt_handle_t . The variable contains the configuration information on a specified AON WDT. |
| Return Value | HAL status |
| Remarks | |

2.26.4.4 hal_aon_wdt_irq_handler

Table 2-449 hal_aon_wdt_irq_handler API

| | |
|-----------------------------|---|
| Function Prototype | void hal_aon_wdt_irq_handler(aon_wdt_handle_t *p_aon_wdt) |
| Function Description | Handle AON WDT interrupt requests. |
| Parameter | p_aon_wdt: pointer to the variables of aon_wdt_handle_t . The variable contains the configuration information on a specified AON WDT. |
| Return Value | None |
| Remarks | |

2.26.4.5 hal_aon_wdt_alarm_callback

Table 2-450 hal_aon_wdt_alarm_callback API

| | |
|-----------------------------|---|
| Function Prototype | void hal_aon_wdt_alarm_callback(aon_wdt_handle_t *p_aon_wdt) |
| Function Description | An interrupt callback which is called when AON WDT counts to alarm_counter |
| Parameter | p_aon_wdt: pointer to the variables of aon_wdt_handle_t . The variable contains the configuration information on a specified AON WDT. |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite this function before using it. |

2.27 HAL WDT Generic Driver

2.27.1 WDT Driver Functionalities

The HAL Watchdog Timer (WDT) driver features the following functionalities:

- Enable and disable the reset mode. When the reset mode is enabled, WDT triggers an interrupt when counting down to 0. In WDT mode, WDT triggers an interrupt when counting down to 0 for the first time, and resets the system when counting down to 0 for the second time.
- Reload the initial value of counting, which means to feed the watchdog.
- Interrupt callback function

2.27.2 How to Use WDT Driver

Developers can use WDT driver in the following scenarios:

1. Overwrite `hal_wdt_msp_init()`, and call `hal_nvic_set_priority()` and `hal_nvic_enable_irq()` in the API to enable NVIC interrupt for WDT.
2. Declare a structure variable of `wdt_handle_t` handle, for example: `wdt_handle_t hwdt`, and set the `p_instance` member as WDT instances.
3. Configure the initial count and the reset mode for the initialization structure of `hwdt` handle.
4. Initialize WDT peripheral by calling `hal_wdt_init()`.
5. If the reset mode of the initialization structure is set to `WDT_RESET_ENABLE`, developers shall reload the initial count by calling `hal_wdt_refresh()` before WDT counts down to 0 for the second time, or the system will be reset by WDT automatically.
6. When WDT counts down to 0 for the first time, the interrupt callback `hal_wdt_period_elapsed_callback()` will be called. Developers can overwrite the API if necessary.

2.27.3 WDT Driver Structures

2.27.3.1 wdt_init_t

The initialization structure `wdt_init_t` of WDT driver is defined below:

Table 2-451 wdt_init_t structure

| Data Field | Field Description | Value |
|----------------------------------|-------------------|---|
| <code>uint32_t counter</code> | Initial count | 0x0000_0000 to 0xFFFF_FFFF |
| <code>uint32_t reset_mode</code> | Counting mode | This parameter can be one of the following values: <ul style="list-style-type: none"> • <code>WDT_RESET_DISABLE</code> (disable reset mode) • <code>WDT_RESET_ENABLE</code> (enable reset mode) |

2.27.3.2 wdt_handle_t

The `wdt_handle_t` structure of WDT driver is defined below:

Table 2-452 wdt_handle_t structure

| Data Field | Field Description | Value |
|-------------------------------------|--|--|
| <code>wdt_regs_t *p_instance</code> | WDT peripheral instance | Value generated with WDT |
| <code>wdt_init_t init</code> | Initialization structure | See " Section 2.27.3.1 wdt_init_t ". |
| <code>__IO hal_lock_t lock</code> | WDT lock (initialization by developers not required) | N/A |

2.27.4 WDT Driver APIs

The WDT driver APIs are listed in the table below:

Table 2-453 WDT driver APIs

| API Type | API Name | Description |
|---------------------------------|---------------------------------|--|
| Initialization | hal_wdt_init() | Initialize WDT peripheral, and configure the initial count and other parameters. |
| | hal_wdt_deinit() | Deinitialize WDT peripheral. |
| | hal_wdt_msp_init() | Initialize NVIC interrupts of WDT. |
| | hal_wdt_msp_deinit() | Deinitialize NVIC interrupts of WDT. |
| I/O operation | hal_wdt_refresh | Reload initial count. |
| Interrupt handling and callback | hal_wdt_irq_handler | Interrupt handler |
| | hal_wdt_period_elapsed_callback | Counting complete callback function |

The sections below elaborate on these APIs.

2.27.4.1 hal_wdt_init

Table 2-454 hal_wdt_init API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_wdt_init(wdt_handle_t *p_wdt) |
| Function Description | Initialize WDT according to specified parameters of WDT handle. |
| Parameter | p_wdt: pointer to the variables of wdt_handle_t . The variable contains the configuration information on a specified WDT. |
| Return Value | HAL status |
| Remarks | |

2.27.4.2 hal_wdt_deinit

Table 2-455 hal_wdt_deinit API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_wdt_deinit(wdt_handle_t *p_wdt) |
| Function Description | Deinitialize the WDT peripheral registers to default reset values. |
| Parameter | p_wdt: pointer to the variables of wdt_handle_t . The variable contains the configuration information on a specified WDT. |
| Return Value | HAL status |
| Remarks | |

2.27.4.3 hal_wdt_msp_init

Table 2-456 hal_wdt_msp_init API

| | |
|---------------------------|--|
| Function Prototype | void hal_wdt_msp_init(wdt_handle_t *p_wdt) |
|---------------------------|--|

| | |
|-----------------------------|---|
| Function Description | Initialize NVIC interrupts of WDT. |
| Parameter | p_wdt: pointer to the variables of wdt_handle_t . The variable contains the configuration information on a specified WDT. |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite the API to initialize the NVIC interrupts. |

2.27.4.4 hal_wdt_msp_deinit

Table 2-457 hal_wdt_msp_deinit API

| | |
|-----------------------------|---|
| Function Prototype | void hal_wdt_msp_deinit(wdt_handle_t *p_wdt) |
| Function Description | Deinitialize NVIC interrupts of WDT. |
| Parameter | p_wdt: pointer to the variables of wdt_handle_t . The variable contains the configuration information on a specified WDT. |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite the API to deinitialize the NVIC interrupts. |

2.27.4.5 hal_wdt_refresh

Table 2-458 hal_wdt_refresh API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_wdt_refresh(wdt_handle_t *p_wdt) |
| Function Description | Refresh WDT counts. |
| Parameter | p_wdt: pointer to the variables of wdt_handle_t . The variable contains the configuration information on a specified WDT. |
| Return Value | HAL status |
| Remarks | |

2.27.4.6 hal_wdt_irq_handler

Table 2-459 hal_wdt_irq_handler API

| | |
|-----------------------------|---|
| Function Prototype | void hal_wdt_irq_handler(wdt_handle_t *p_wdt) |
| Function Description | Handle WDT interrupt requests. |
| Parameter | p_wdt: pointer to the variables of wdt_handle_t . The variable contains the configuration information on a specified WDT. |
| Return Value | None |

| | |
|---------|--|
| Remarks | |
|---------|--|

2.27.4.7 hal_wdt_period_elapsed_callback

Table 2-460 hal_wdt_period_elapsed_callback API

| | |
|-----------------------------|---|
| Function Prototype | void hal_wdt_period_elapsed_callback(wdt_handle_t *p_wdt) |
| Function Description | The interrupt callback function that enables WDT to count down to 0 |
| Parameter | p_wdt: pointer to the variables of wdt_handle_t . The variable contains the configuration information on a specified WDT. |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite this function before using it. |

2.28 HAL COMP Generic Driver

2.28.1 COMP Driver Functionalities

The HAL comparator (COMP) driver features the following functionalities:

- Configurable input source and reference source
- Result interrupt trigger and interrupt callback function

2.28.2 How to Use COMP Driver

Developers can use the COMP driver in the following scenarios:

1. Overwrite hal_comp_msp_init(), and call hal_nvic_set_priority() and hal_nvic_enable_irq() in the API to enable NVIC interrupt for COMP.
2. Overwrite hal_comp_trigger_callback().
3. Declare a structure variable of comp_handle_t, for example: comp_handle_t g_comp_handle.
4. Configure the input source, reference source, and reference value for the initialization structure of g_comp_handle.
5. Initialize COMP module by calling hal_comp_init().
6. Start the comparator by calling hal_comp_start.
7. Stop the comparator by calling hal_comp_stop.

Note:

- GR5515 SoCs operate with single power supplies, and do not support negative voltage input.
- In practice, use external circuits as inputs to avoid I/O floating.

2.28.3 COMP Driver Structures

2.28.3.1 comp_init_t

The initialization structure `comp_init_t` of COMP driver is defined below:

```
typedef ll_comp_init_t comp_init_t
```

For more information, see "[Section 3.17.1.1 ll_comp_init_t](#)".

2.28.3.2 comp_handle_t

The handle structure `comp_handle_t` of COMP driver is defined below:

Table 2-461 `comp_handle_t` structure

| Data Field | Field Description | Value |
|--|--|---|
| <code>comp_init_t init</code> | Initialization structure | See " Section 2.28.3.1 comp_init_t ". |
| <code>__IO hal_lock_t lock</code> | COMP lock (initialization by developers not required) | N/A |
| <code>__IO hal_comp_state_t state</code> | COMP state | This parameter can be one of the following values: <ul style="list-style-type: none"> • HAL_COMP_STATE_RESET (not initialized) • HAL_COMP_STATE_READY (initialized and ready for use) • HAL_COMP_STATE_BUSY (busy) • HAL_COMP_STATE_ERROR (error) |
| <code>__IO uint32_t error_code</code> | COMP error code | This parameter can be one of the following values: <ul style="list-style-type: none"> • HAL_COMP_ERROR_NONE • HAL_COMP_ERROR_TIMEOUT • HAL_COMP_ERROR_INVALID_PARAM |
| <code>uint32_t retention[1]</code> | COMP register information (managed by COMP driver and initialization by developers not required) | N/A |

2.28.4 COMP Driver APIs

The COMP driver APIs are listed in the table below:

Table 2-462 COMP driver APIs

| API Type | API Name | Description |
|---------------------------------|-----------------------------|--|
| Initialization | hal_comp_init() | Initialize COMP module; configure input source, reference source, reference value, and other parameters; call hal_comp_msp_init to make configuration for interrupt; initialize COMP state and error code. |
| | hal_comp_deinit() | Deinitialize COMP registers, COMP state, and error code. |
| | hal_comp_msp_init() | Initialize NVIC interrupts of COMP. |
| | hal_comp_msp_deinit() | Deinitialize NVIC interrupts of COMP. |
| I/O operation | hal_comp_start() | Start the comparator. |
| | hal_comp_stop() | Stop the comparator. |
| Interrupt handling and callback | hal_comp_irq_handler() | Interrupt handler |
| | hal_comp_trigger_callback() | Interrupt callback function |
| State and error | hal_comp_get_state() | Get the driver operating state. |
| | hal_comp_get_error() | Get error code. |
| Sleep | hal_comp_suspend_reg() | Suspend registers related to COMP configuration before the device enters sleep mode. |
| | hal_comp_resume_reg() | Resume registers related to COMP configuration during wakeup. |

The sections below elaborate on these APIs.

2.28.4.1 hal_comp_init

Table 2-463 hal_comp_init API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_comp_init(comp_handle_t *p_comp) |
| Function Description | Initialize COMP module according to the configuration parameters in a specified COMP handle. |
| Parameter | p_comp: pointer to the variables of comp_handle_t . The variable contains the configuration information on a specified COMP. |
| Return Value | HAL status |
| Remarks | |

2.28.4.2 hal_comp_deinit

Table 2-464 hal_comp_deinit API

| | |
|-----------------------------|---|
| Function Prototype | hal_status_t hal_comp_deinit(comp_handle_t *p_comp) |
| Function Description | Deinitialize COMP registers to their default reset values; reset the status and error code of the comparator. |

| | |
|---------------------|--|
| Parameter | p_comp: pointer to the variables of comp_handle_t . The variable contains the configuration information on a specified COMP. |
| Return Value | HAL status |
| Remarks | |

2.28.4.3 hal_comp_msp_init

Table 2-465 hal_comp_msp_init API

| | |
|-----------------------------|--|
| Function Prototype | void hal_comp_msp_init(comp_handle_t *p_comp) |
| Function Description | Initialize NVIC interrupts of COMP. |
| Parameter | p_comp: pointer to the variables of comp_handle_t . The variable contains the configuration information on a specified COMP. |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite the API to enable relevant functionalities. |

2.28.4.4 hal_comp_msp_deinit()

Table 2-466 hal_comp_msp_deinit API

| | |
|-----------------------------|--|
| Function Prototype | void hal_comp_msp_deinit(comp_handle_t *p_comp) |
| Function Description | Deinitialize NVIC interrupts of COMP. |
| Parameter | p_comp: pointer to the variables of comp_handle_t . The variable contains the configuration information on a specified COMP. |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite the API to enable relevant functionalities. |

2.28.4.5 hal_comp_start

Table 2-467 hal_comp_start API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_comp_start(comp_handle_t *p_comp) |
| Function Description | Start the comparator. When the input voltage exceeds the reference voltage, COMP interrupt is triggered. |
| Parameter | p_comp: pointer to the variables of comp_handle_t . The variable contains the configuration information on a specified COMP. |
| Return Value | HAL status |
| Remarks | |

2.28.4.6 hal_comp_stop

Table 2-468 hal_comp_stop API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_comp_stop(comp_handle_t *p_comp) |
| Function Description | Stop the comparator. |
| Parameter | p_comp: pointer to the variables of comp_handle_t . The variable contains the configuration information on a specified COMP. |
| Return Value | HAL status |
| Remarks | |

2.28.4.7 hal_comp_irq_handler

Table 2-469 hal_comp_irq_handler API

| | |
|-----------------------------|--|
| Function Prototype | void hal_comp_irq_handler(comp_handle_t *p_comp) |
| Function Description | Handle COMP interrupt requests. |
| Parameter | p_comp: pointer to the variables of comp_handle_t . The variable contains the configuration information on a specified COMP. |
| Return Value | None |
| Remarks | |

2.28.4.8 hal_comp_trigger_callback

Table 2-470 hal_comp_trigger_callback API

| | |
|-----------------------------|--|
| Function Prototype | void hal_comp_trigger_callback(comp_handle_t *p_comp) |
| Function Description | Interrupt callback function |
| Parameter | p_comp: pointer to the variables of comp_handle_t . The variable contains the configuration information on a specified COMP. |
| Return Value | None |
| Remarks | The function is declared empty as weak function. Developers are required to overwrite this function before using it. |

2.28.4.9 hal_comp_get_state

Table 2-471 hal_comp_get_state API

| | |
|-----------------------------|--|
| Function Prototype | hal_comp_state_t hal_comp_get_state(comp_handle_t *p_comp) |
| Function Description | Get the operating state of the comparator. |

| | |
|---------------------|--|
| Parameter | p_comp: pointer to the variables of comp_handle_t . The variable contains the configuration information on a specified COMP. |
| Return Value | COMP state. The parameter can be one of the following values: <ul style="list-style-type: none"> • HAL_COMP_STATE_RESET (not initialized) • HAL_COMP_STATE_READY (initialized and ready for use) • HAL_COMP_STATE_BUSY (busy) • HAL_COMP_STATE_ERROR (error) |
| Remarks | |

2.28.4.10 hal_comp_get_error

Table 2-472 hal_comp_get_error API

| | |
|-----------------------------|--|
| Function Prototype | uint32_t hal_comp_get_error(comp_handle_t *p_comp) |
| Function Description | Get COMP error code. |
| Parameter | p_comp: pointer to the variables of comp_handle_t . The variable contains the configuration information on a specified COMP. |
| Return Value | COMP error code. The parameter can be one of the following values: <ul style="list-style-type: none"> • HAL_COMP_ERROR_NONE • HAL_COMP_ERROR_TIMEOUT • HAL_COMP_ERROR_INVALID_PARAM |
| Remarks | |

2.28.4.11 hal_comp_suspend_reg

Table 2-473 hal_comp_suspend_reg API

| | |
|-----------------------------|--|
| Function Prototype | hal_status_t hal_comp_suspend_reg(comp_handle_t *p_comp); |
| Function Description | Suspend registers related to COMP configuration before the device enters sleep mode. |
| Parameter | p_comp: pointer to the variables of comp_handle_t . The variable contains the configuration information on a specified COMP. |
| Return Value | HAL status |
| Remarks | |

2.28.4.12 hal_comp_resume_reg

Table 2-474 hal_comp_resume_reg API

| | |
|---------------------------|--|
| Function Prototype | hal_status_t hal_comp_resume_reg(comp_handle_t *p_comp); |
|---------------------------|--|

| | |
|-----------------------------|--|
| Function Description | Resume registers related to COMP configuration during wakeup. |
| Parameter | p_comp: pointer to the variables of comp_handle_t . The variable contains the configuration information on a specified COMP. |
| Return Value | HAL status |
| Remarks | |

3 LL Drivers

3.1 Introduction

This section introduces common LL driver resources of each module and methods on how to use LL drivers.

Note:

This chapter focuses on initialization APIs of the LL drivers. For information about more APIs, see *GR551x API Reference*.

3.1.1 LL Common Resources

For LL drivers of GR551x SoCs, the common resources of all peripherals including common enumerations, structures, and macros are defined in *gr55xx.h*. The details are as follows:

1. Flag status/Interrupt status: showing whether the relevant flags or interrupt flags are set to 1. Definition:

```
typedef enum
{
    RESET = 0,
    SET = !RESET
} flag_status, it_status;
```

2. Functional status: showing whether the relevant functionalities are enabled. Definition:

```
typedef enum
{
    DISABLE = 0,
    ENABLE = !DISABLE
} functional_state;
```

3. Common macros: mainly relevant to registers, enabling direct read/write and bitwise read/write of registers. Definition:

```
#define SET_BIT(REG, BIT)      ((REG) |= (BIT))
#define CLEAR_BIT(REG, BIT)   ((REG) &= ~(BIT))
#define READ_BIT(REG, BIT)    ((REG) & (BIT))

#define CLEAR_REG(REG)        ((REG) = (0x0))
#define WRITE_REG(REG, VAL)   ((REG) = (VAL))
#define READ_REG(REG)         ((REG))
#define MODIFY_REG(REG, CLEARMASK, SETMASK)  WRITE_REG((REG), ((READ_REG(REG)) & (~CLEARMASK)) | (SETMASK))

#define POSITION_VAL(VAL)      (__CLZ(__RBIT(VAL)))
```

3.1.2 How to Use LL Drivers

LL drivers provide APIs for peripheral registers. The APIs shall be used by following the rules below:

1. If the LL driver of a peripheral provides the initialization API `ll_ppp_init()`, users shall first initialize the peripheral by calling the API.

2. Call the relevant APIs based on required functionalities, and make corresponding operations.
3. If the LL driver of a peripheral provides the deinitialization API `ll_ppp_deinit()`, users can deinitialize the peripheral by calling the API.

3.2 LL GPIO Generic Driver

3.2.1 GPIO Driver Structure

3.2.1.1 ll_gpio_init_t

The initialization structure `ll_gpio_init_t` of the LL GPIO driver is defined below:

Table 3-1 ll_gpio_init_t structure

| Data Field | Field Description | Value |
|----------------------------|---|---|
| <code>uint32_t pin</code> | Specify the GPIO pins to be configured. | <p>This parameter can be any combination of the following values:</p> <ul style="list-style-type: none"> • <code>LL_GPIO_PIN_0</code> (Pin 0) • <code>LL_GPIO_PIN_1</code> (Pin 1) • <code>LL_GPIO_PIN_2</code> (Pin 2) • <code>LL_GPIO_PIN_3</code> (Pin 3) • <code>LL_GPIO_PIN_4</code> (Pin 4) • <code>LL_GPIO_PIN_5</code> (Pin 5) • <code>LL_GPIO_PIN_6</code> (Pin 6) • <code>LL_GPIO_PIN_7</code> (Pin 7) • <code>LL_GPIO_PIN_8</code> (Pin 8) • <code>LL_GPIO_PIN_9</code> (Pin 9) • <code>LL_GPIO_PIN_10</code> (Pin 10) • <code>LL_GPIO_PIN_11</code> (Pin 11) • <code>LL_GPIO_PIN_12</code> (Pin 12) • <code>LL_GPIO_PIN_13</code> (Pin 13) • <code>LL_GPIO_PIN_14</code> (Pin 14) • <code>LL_GPIO_PIN_15</code> (Pin 15) • <code>LL_GPIO_PIN_ALL</code> (all pins) |
| <code>uint32_t mode</code> | <p>Specify the operating mode of the selected pin.</p> <p>Developers can also set the parameter by running <code>ll_gpio_set_pin_mode()</code>.</p> | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • <code>LL_GPIO_MODE_INPUT</code> (input mode) • <code>LL_GPIO_MODE_OUTPUT</code> (output mode) • <code>LL_GPIO_MODE_MUX</code> (multiplexing mode) |

| Data Field | Field Description | Value |
|------------------|---|---|
| uint32_t pull | Specify the type of pull-up/pull-down resistors of the selected pin. Developers can also set the parameter by running ll_gpio_set_pin_pull(). | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> LL_GPIO_PULL_NO (no pull-up/pull-down resistor activated) LL_GPIO_PULL_UP (activate pull-up resistor) LL_GPIO_PULL_DOWN (activate pull-down resistor) |
| uint32_t mux | Specify the multiplexing mode of the selected pin. Developers can also set the parameter by running ll_gpio_set_mux_pin_0_7() and ll_gpio_set_mux_pin_8_15(). | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> LL_GPIO_MUX_0 (multiplexing mode 0) LL_GPIO_MUX_1 (multiplexing mode 1) LL_GPIO_MUX_2 (multiplexing mode 2) LL_GPIO_MUX_3 (multiplexing mode 3) LL_GPIO_MUX_4 (multiplexing mode 4) LL_GPIO_MUX_5 (multiplexing mode 5) LL_GPIO_MUX_6 (multiplexing mode 6) LL_GPIO_MUX_7 (multiplexing mode 7) |
| uint32_t trigger | Specify the interrupt triggering mode of the selected pin. Developers can also set the parameter by running ll_gpio_enable_falling_trig(), ll_gpio_enable_rising_trig(), ll_gpio_enable_high_trig(), and ll_gpio_enable_low_trig(). | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> LL_AON_GPIO_TRIGGER_NONE (no interrupt triggered) LL_AON_GPIO_TRIGGER_RISING (interrupt triggered by rising edge) LL_AON_GPIO_TRIGGER_FALLING (interrupt triggered by falling edge) LL_AON_GPIO_TRIGGER_HIGH (interrupt triggered at high voltage level) LL_AON_GPIO_TRIGGER_LOW (interrupt triggered at low voltage level) |

3.2.2 GPIO Driver APIs

The GPIO driver APIs are listed in the table below:

Table 3-2 GPIO driver APIs

| API Type | API Name | Description |
|---------------------------------|-----------------------|--|
| Initialization/Deinitialization | ll_gpio_init() | Initialize GPIO peripheral. |
| | ll_gpio_deinit() | Deinitialize GPIO peripheral to default. |
| | ll_gpio_struct_init() | Initialize variables of ll_gpio_init_t to default. |

The sections below elaborate on these APIs.

3.2.2.1 ll_gpio_init

Table 3-3 ll_gpio_init API

| | |
|-----------------------------|---|
| Function Prototype | error_status_t ll_gpio_init(gpio_regs_t *GPIOx, ll_gpio_init_t *p_gpio_init) |
| Function Description | Initialize GPIO peripheral based on specified parameters in ll_gpio_init_t . |
| Parameter | GPIOx: x can be 0 or 1, which specifies the GPIO port in use in the GR551x family p_gpio_init: pointer to variables of ll_gpio_init_t . The variable contains the configuration information of a specified GPIO peripheral instance. |
| Return Value | error_status_t shows the enumeration type, which can be: <ul style="list-style-type: none"> • SUCCESS: Initialization of GPIO peripheral registers succeeds. • ERROR: N/A |
| Remarks | |

3.2.2.2 ll_gpio_deinit

Table 3-4 ll_gpio_deinit API

| | |
|-----------------------------|--|
| Function Prototype | error_status_t ll_gpio_deinit(gpio_regs_t *GPIOx) |
| Function Description | Deinitialize the GPIO peripheral registers to default reset values. |
| Parameter | GPIOx: x can be 0 or 1, which specifies the GPIO port in use in the GR551x family |
| Return Value | error_status_t shows the enumeration type, which can be: <ul style="list-style-type: none"> • SUCCESS: Deinitialization of GPIO peripheral registers succeeds. • ERROR: Errors occur in GPIOx variables. |
| Remarks | |

3.2.2.3 ll_gpio_struct_init

Table 3-5 ll_gpio_struct_init API

| | |
|-----------------------------|---|
| Function Prototype | void ll_gpio_struct_init(ll_gpio_init_t *p_gpio_init) |
| Function Description | Initialize variables of ll_gpio_init_t to default values. |
| Parameter | p_gpio_init: pointer to structure variables to be reset |
| Return Value | None |
| Remarks | |

3.3 LL AON GPIO Generic Driver

3.3.1 AON GPIO Driver Structure

3.3.1.1 ll_aon_gpio_init_t

The initialization structure `ll_aon_gpio_init_t` of the LL AON GPIO driver is defined below:

Table 3-6 `ll_aon_gpio_init_t` structure

| Data Field | Field Description | Value |
|-------------------------------|---|---|
| <code>uint32_t pin</code> | AON GPIO pins to be configured | <p>This parameter can be any combination of the following values:</p> <ul style="list-style-type: none"> LL_AON_GPIO_PIN_0 (Pin 0) LL_AON_GPIO_PIN_1 (Pin 1) LL_AON_GPIO_PIN_2 (Pin 2) LL_AON_GPIO_PIN_3 (Pin 3) LL_AON_GPIO_PIN_4 (Pin 4) LL_AON_GPIO_PIN_5 (Pin 5) LL_AON_GPIO_PIN_6 (Pin 6) LL_AON_GPIO_PIN_7 (Pin 7) LL_AON_GPIO_PIN_ALL (all pins) |
| <code>uint32_t mode</code> | The designated operating mode of the selected pin. Developers can also set the parameter by running <code>ll_aon_gpio_set_pin_mode()</code> . | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> LL_AON_GPIO_MODE_INPUT (input mode) LL_AON_GPIO_MODE_OUTPUT (output mode) LL_AON_GPIO_MODE_MUX (multiplexing mode) |
| <code>uint32_t pull</code> | The pull-up/pull-down resistors of the selected pin. Developers can also set the parameter by running <code>ll_aon_gpio_set_pin_pull()</code> . | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> LL_AON_GPIO_PULL_NO (pull-up/pull-down resistor not activated) LL_AON_GPIO_PULL_UP (activate pull-up resistor) LL_AON_GPIO_PULL_DOWN (activate pull-down resistor) |
| <code>uint32_t trigger</code> | The interrupt triggering mode of the selected pin. Developers can also set the parameter by running <code>ll_aon_gpio_enable_falling_trig()</code> , <code>ll_aon_gpio_enable_rising_trig()</code> , <code>ll_aon_gpio_enable_high_trig()</code> , and <code>ll_aon_gpio_enable_low_trig()</code> . | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> LL_AON_GPIO_TRIGGER_NONE (no interrupt triggered) LL_AON_GPIO_TRIGGER_RISING (interrupt triggered by rising edge) LL_AON_GPIO_TRIGGER_FALLING (interrupt triggered by falling edge) LL_AON_GPIO_TRIGGER_HIGH (interrupt triggered at high voltage level) LL_AON_GPIO_TRIGGER_LOW (interrupt triggered at low voltage level) |

3.3.2 AON GPIO Driver APIs

The AON GPIO driver APIs are listed in the table below:

Table 3-7 AON GPIO driver APIs

| API Type | API Name | Description |
|---------------------------------|---------------------------|--|
| Initialization/Deinitialization | ll_aon_gpio_init() | Initialize AON GPIO peripheral. |
| | ll_aon_gpio_deinit() | Deinitialize AON GPIO peripheral to default. |
| | ll_aon_gpio_struct_init() | Initialize the structure aon_gpio_init to default. |

The sections below elaborate on these APIs.

3.3.2.1 ll_aon_gpio_init

Table 3-8 ll_aon_gpio_init API

| | |
|-----------------------------|---|
| Function Prototype | error_status ll_aon_gpio_init(ll_aon_gpio_init_t *p_aon_gpio_init) |
| Function Description | Initialize AON GPIO peripheral based on specified parameters in ll_aon_gpio_init_t . |
| Parameter | p_aon_gpio_init: pointer to variables of ll_aon_gpio_init_t . The variable contains the configuration information of a specified AON GPIO pin. |
| Return Value | error_status_t shows the enumeration type, which can be: <ul style="list-style-type: none"> • SUCCESS: Initialization of GPIO peripheral registers succeeds. • ERROR: Initialization fails. |
| Remarks | |

3.3.2.2 ll_aon_gpio_deinit

Table 3-9 ll_aon_gpio_deinit API

| | |
|-----------------------------|---|
| Function Prototype | error_status_t ll_aon_gpio_deinit(void) |
| Function Description | Deinitialize the GPIO peripheral registers to default reset values. |
| Parameter | None |
| Return Value | error_status_t shows the enumeration type, which can be: <ul style="list-style-type: none"> • SUCCESS: Deinitialization of AON GPIO peripheral registers succeeds. • ERROR: Deinitialization fails. |
| Remarks | |

3.3.2.3 ll_aon_gpio_struct_init

Table 3-10 ll_aon_gpio_struct_init API

| | |
|-----------------------------|---|
| Function Prototype | void ll_aon_gpio_struct_init(ll_aon_gpio_init_t *p_aon_gpio_init) |
| Function Description | Initialize variables of ll_aon_gpio_init_t to default reset values. |
| Parameter | p_aon_gpio_init: pointer to structure variables to be reset |

| | |
|--------------|------|
| Return Value | None |
| Remarks | |

3.4 LL ADC Generic Driver

3.4.1 ADC Driver Structure

3.4.1.1 ll_adc_init_t

The initialization structure ll_adc_init_t of the LL ADC driver is defined below:

Table 3-11 ll_adc_init_t structure

| Data Field | Field Description | Value |
|---------------------|-----------------------|--|
| uint32_t channel_p | Input for Channel P | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • LL_ADC_INPUT_SRC_IO0 (MSIO 0 input) • LL_ADC_INPUT_SRC_IO1 (MSIO 1 input) • LL_ADC_INPUT_SRC_IO2 (MSIO 2 input) • LL_ADC_INPUT_SRC_IO3 (MSIO 3 input) • LL_ADC_INPUT_SRC_IO4 (MSIO 4 input) • LL_ADC_INPUT_SRC_TMP (temperature sensor input) • LL_ADC_INPUT_SRC_BAT (battery voltage input) • LL_ADC_INPUT_SRC_REF (reference voltage input) |
| uint32_t channel_n | Input for Channel N | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • LL_ADC_INPUT_SRC_IO0 (MSIO 0 input) • LL_ADC_INPUT_SRC_IO1 (MSIO 1 input) • LL_ADC_INPUT_SRC_IO2 (MSIO 2 input) • LL_ADC_INPUT_SRC_IO3 (MSIO 3 input) • LL_ADC_INPUT_SRC_IO4 (MSIO 4 input) • LL_ADC_INPUT_SRC_TMP (temperature sensor input) • LL_ADC_INPUT_SRC_BAT (battery voltage input) • LL_ADC_INPUT_SRC_REF (reference voltage input) |
| uint32_t input_mode | Sampling mode | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • LL_ADC_INPUT_SINGLE (single-ended input mode) • LL_ADC_INPUT_DIFFERENTIAL (differential input mode) |
| uint32_t ref_source | Reference source type | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • LL_ADC_REF_SRC_BUF_INT (internal buffered reference source) |

| Data Field | Field Description | Value |
|-------------------|----------------------------|--|
| | | <ul style="list-style-type: none"> LL_ADC_REF_SRC_IO0 (MSIO 0 input voltage) LL_ADC_REF_SRC_IO1 (MSIO 1 input voltage) LL_ADC_REF_SRC_IO2 (MSIO 2 input voltage) LL_ADC_REF_SRC_IO3 (MSIO 3 input voltage) |
| uin32_t ref_value | Internal reference voltage | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> LL_ADC_REF_VALUE_0P8 (0.85 V) LL_ADC_REF_VALUE_1P2 (1.28 V) LL_ADC_REF_VALUE_1P6 (1.6 V) <p>Note: The external input signal range: 0 to (2 x ref_value). You can set this value based on actual requirements.</p> |
| uin32_t clock | Sampling clock | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> LL_ADC_CLK_16M (16 MHz clock) LL_ADC_CLK_1P6M (1.6 MHz clock) LL_ADC_CLK_8M (8 MHz clock) LL_ADC_CLK_4M (4 MHz clock) LL_ADC_CLK_2M (2 MHz clock) LL_ADC_CLK_1M (1 MHz clock) |

3.4.2 ADC Driver APIs

The ADC driver APIs are listed in the table below:

Table 3-12 ADC driver APIs

| API Type | API Name | Description |
|---------------------------------|----------------------|--|
| Initialization/Deinitialization | ll_adc_init() | Initialize ADC peripheral. |
| | ll_adc_deinit() | Deinitialize ADC peripheral to default. |
| | ll_adc_struct_init() | Initialize the structure ll_adc_init_t to default. |

The sections below elaborate on these APIs.

3.4.2.1 ll_adc_init

Table 3-13 ll_adc_init API

| | |
|-----------------------------|---|
| Function Prototype | error_status_t ll_adc_init(ll_adc_init_t *p_adc_init) |
| Function Description | Initialize ADC peripheral based on specified parameters in ll_adc_init_t. |

| | |
|---------------------|--|
| Parameter | p_adc_init: pointer to variables of ll_adc_init_t. The variable contains the configuration information of a specified ADC pin. |
| Return Value | error_status_t shows the enumeration type, which can be: <ul style="list-style-type: none"> • SUCCESS: Initialization of ADC peripheral registers succeeds. • ERROR: Initialization fails. |
| Remarks | |

3.4.2.2 ll_adc_deinit

Table 3-14 ll_adc_deinit API

| | |
|-----------------------------|--|
| Function Prototype | error_status_t ll_adc_deinit(void) |
| Function Description | Deinitialize ADC peripheral registers and set the register values as default. |
| Parameter | None |
| Return Value | error_status_t shows the enumeration type, which can be: <ul style="list-style-type: none"> • SUCCESS: Deinitialization of ADC peripheral registers succeeds. • ERROR: Deinitialization fails. |
| Remarks | |

3.4.2.3 ll_adc_struct_init

Table 3-15 ll_adc_struct_init API

| | |
|-----------------------------|--|
| Function Prototype | void ll_adc_struct_init(ll_adc_init_t *p_adc_init) |
| Function Description | Initialize variables of ll_adc_init_t to default values. |
| Parameter | p_adc_init: pointer to structure variables to be reset |
| Return Value | None |
| Remarks | |

3.5 LL DMA Generic Driver

3.5.1 DMA Driver Structure

3.5.1.1 ll_dma_init_t

The initialization structure ll_dma_init_t of the LL DMA driver is defined below:

Table 3-16 ll_dma_init_t structure

| Data Field | Field Description | Value |
|-----------------------------|--|--|
| uint32_t src_address | Source address, which can also be set with ll_dma_set_source_address() | 0x0000_0000 to 0xFFFF_FFFF |
| uint32_t dst_address | Destination address, which can also be set with ll_dma_set_destination_address() | 0x0000_0000 to 0xFFFF_FFFF |
| uint32_t direction | Transfer direction, which can also be set with ll_dma_set_data_transfer_direction() | This parameter can be one of the following values: <ul style="list-style-type: none"> LL_DMA_DIRECTION_MEMORY_TO_MEMORY (memory to memory) LL_DMA_DIRECTION_MEMORY_TO_PERIPH (memory to peripheral) LL_DMA_DIRECTION_PERIPH_TO_MEMORY (peripheral to memory) LL_DMA_DIRECTION_PERIPH_TO_PERIPH (peripheral to peripheral) |
| uint32_t mode | Transfer mode, which can also be set with ll_dma_set_mode() after initialization | This parameter can be one of the following values: <ul style="list-style-type: none"> LL_DMA_MODE_SINGLE_BLOCK (single-block transfer) LL_DMA_MODE_MULTI_BLOCK_SRC_RELOAD (multiple-block transfer, automatic reload of source address) LL_DMA_MODE_MULTI_BLOCK_DST_RELOAD (multiple-block transfer, automatic reload of destination address) LL_DMA_MODE_MULTI_BLOCK_ALL_RELOAD (automatic reload of multiple-block transfer, source address and destination address) |
| uint32_t src_increment_mode | Increment mode for source address, which can also be set the parameter with ll_dma_set_mode() after initialization | This parameter can be one of the following values: <ul style="list-style-type: none"> LL_DMA_SRC_INCREMENT (source address increments) LL_DMA_SRC_DECREMENT (source address decrements) LL_DMA_SRC_NO_CHANGE (source address remains unchanged) |
| uint32_t dst_increment_mode | Increment mode for destination address, which can also be set with ll_dma_set_mode() after initialization | This parameter can be one of the following values: <ul style="list-style-type: none"> LL_DMA_DST_INCREMENT (destination address increments) LL_DMA_DST_DECREMENT (destination address decrements) |

| Data Field | Field Description | Value |
|-------------------------|--|--|
| | | <ul style="list-style-type: none"> LL_DMA_DST_NO_CHANGE (destination address remains unchanged) |
| uint32_t src_data_width | Width of source data transmitted in burst mode, which can also be set with ll_dma_set_mode() after initialization | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> LL_DMA_SRC_BURST_LENGTH_1 (1 byte) LL_DMA_SRC_BURST_LENGTH_4 (4 bytes) LL_DMA_SRC_BURST_LENGTH_8 (8 bytes) |
| uint32_t dst_data_width | Width of destination data transmitted in burst mode, which can also be set with ll_dma_set_mode() after initialization | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> LL_DMA_DST_BURST_LENGTH_1 (1 byte) LL_DMA_DST_BURST_LENGTH_4 (4 bytes) LL_DMA_DST_BURST_LENGTH_8 (8 bytes) |
| uint32_t block_size | Data TX size, which can also be set with ll_dma_set_block_size() after initialization | 0 to 4095 |
| uint32_t src_peripheral | Source peripheral, which can also be set with ll_dma_set_source_peripheral() after initialization | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> LL_DMA_PERIPH_SPIM_TX (SPIM TX) LL_DMA_PERIPH_SPIM_RX (SPIM RX) LL_DMA_PERIPH_SPIS_TX (SPIS TX) LL_DMA_PERIPH_SPIS_RX (SPIS RX) LL_DMA_PERIPH_QSPI0_TX (QSPI0 TX) LL_DMA_PERIPH_QSPI0_RX (QSPI0 RX) LL_DMA_PERIPH_I2C0_TX (I2C0 TX) LL_DMA_PERIPH_I2C0_RX (I2C0 RX) LL_DMA_PERIPH_I2C1_TX (I2C1 TX) LL_DMA_PERIPH_I2C1_RX (I2C1 RX) LL_DMA_PERIPH_I2S_S_TX (I2SS TX) LL_DMA_PERIPH_I2S_S_RX (I2SS RX) LL_DMA_PERIPH_UART0_TX (UART0 TX) LL_DMA_PERIPH_UART0_RX (UART0 RX) LL_DMA_PERIPH_QSPI1_TX (QSPI1 TX) LL_DMA_PERIPH_QSPI1_RX (QSPI1 RX) LL_DMA_PERIPH_I2S_M_TX (I2SM TX) LL_DMA_PERIPH_I2S_M_RX (I2SM RX) LL_DMA_PERIPH_SNSADC (Sense ADC) LL_DMA_PERIPH_MEM (memory) |

| Data Field | Field Description | Value |
|-------------------------|---|--|
| uint32_t dst_peripheral | Destination peripheral, which can also be set with <code>ll_dma_set_destination_peripheral()</code> after initialization | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • <code>LL_DMA_PERIPH_SPIM_TX</code> (SPIM TX) • <code>LL_DMA_PERIPH_SPIM_RX</code> (SPIM RX) • <code>LL_DMA_PERIPH_SPIS_TX</code> (SPIS TX) • <code>LL_DMA_PERIPH_SPIS_RX</code> (SPIS RX) • <code>LL_DMA_PERIPH_QSPI0_TX</code> (QSPI0 TX) • <code>LL_DMA_PERIPH_QSPI0_RX</code> (QSPI0 RX) • <code>LL_DMA_PERIPH_I2C0_TX</code> (I2C0 TX) • <code>LL_DMA_PERIPH_I2C0_RX</code> (I2C0 RX) • <code>LL_DMA_PERIPH_I2C1_TX</code> (I2C1 TX) • <code>LL_DMA_PERIPH_I2C1_RX</code> (I2C1 RX) • <code>LL_DMA_PERIPH_UART0_TX</code> (UART0 TX) • <code>LL_DMA_PERIPH_UART0_RX</code> (UART0 RX) • <code>LL_DMA_PERIPH_QSPI1_TX</code> (QSPI1 TX) • <code>LL_DMA_PERIPH_QSPI1_RX</code> (QSPI1 RX) • <code>LL_DMA_PERIPH_SNSADC</code> (Sense ADC) • <code>LL_DMA_PERIPH_MEM</code> (memory) |
| uint32_t priority | Channel priority level, which can also be set with <code>ll_dma_set_channel_priority_level()</code> after initialization. | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • <code>LL_DMA_PRIORITY_0</code> (priority level 0, the lowest) • <code>LL_DMA_PRIORITY_1</code> (priority level 1) • <code>LL_DMA_PRIORITY_2</code> (priority level 2) • <code>LL_DMA_PRIORITY_3</code> (priority level 3) • <code>LL_DMA_PRIORITY_4</code> (priority level 4) • <code>LL_DMA_PRIORITY_5</code> (priority level 5) • <code>LL_DMA_PRIORITY_6</code> (priority level 6) • <code>LL_DMA_PRIORITY_7</code> (priority level 7, the highest) |

3.5.2 DMA Driver APIs

The DMA driver APIs are listed in the table below:

Table 3-17 DMA driver APIs

| API Type | API Name | Description |
|---------------------------------|------------------------------|---|
| Initialization/Deinitialization | <code>ll_dma_init()</code> | Initialize the specified DMA channels. |
| | <code>ll_dma_deinit()</code> | Deinitialize DMA peripheral to default. |

| API Type | API Name | Description |
|----------|----------------------|---|
| | ll_dma_struct_init() | Initialize the structure dma_init to default. |

The sections below elaborate on these APIs.

3.5.2.1 ll_dma_init

Table 3-18 ll_dma_init API

| | |
|-----------------------------|---|
| Function Prototype | error_status_t ll_dma_init(dma_regs_t *DMAx, uint32_t channel, ll_dma_init_t *p_dma_init) |
| Function Description | Initialize a specified DMA channel according to the specified parameters in ll_dma_init_t . |
| Parameter | <p>DMAx: DMA peripheral instance</p> <p>channel: Specify the DMA channel to be initialized. This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • LL_DMA_CHANNEL_0 • LL_DMA_CHANNEL_1 • LL_DMA_CHANNEL_2 • LL_DMA_CHANNEL_3 • LL_DMA_CHANNEL_4 • LL_DMA_CHANNEL_5 • LL_DMA_CHANNEL_6 • LL_DMA_CHANNEL_7 <p>p_dma_init: pointer to the variables of ll_dma_init_t. The variable contains the configuration information on a specified DMA channel.</p> |
| Return Value | <p>error_status_t shows the enumeration type, which can be:</p> <ul style="list-style-type: none"> • SUCCESS: Initialization of DMA peripheral registers succeeds. • ERROR: Initialization fails. |
| Remarks | |

3.5.2.2 ll_dma_deinit

Table 3-19 ll_dma_deinit API

| | |
|-----------------------------|--|
| Function Prototype | error_status_t ll_dma_deinit(dma_regs_t *DMAx, uint32_t channel) |
| Function Description | Deinitialize the registers of specified DMA channels to default reset values. |
| Parameter | <p>DMAx: pointer to DMA peripheral instance</p> <p>channel: Specify the DMA channel to be initialized. This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • LL_DMA_CHANNEL_0 • LL_DMA_CHANNEL_1 |

| | |
|---------------------|---|
| | <ul style="list-style-type: none"> • LL_DMA_CHANNEL_2 • LL_DMA_CHANNEL_3 • LL_DMA_CHANNEL_4 • LL_DMA_CHANNEL_5 • LL_DMA_CHANNEL_6 • LL_DMA_CHANNEL_7 |
| Return Value | <p>error_status_t shows the enumeration type, which can be:</p> <ul style="list-style-type: none"> • SUCCESS: Deinitialization of DMA peripheral registers succeeds. • ERROR: Deinitialization fails. |
| Remarks | |

3.5.2.3 ll_dma_struct_init

Table 3-20 ll_dma_struct_init API

| | |
|-----------------------------|--|
| Function Prototype | void ll_dma_struct_init(ll_dma_init_t *p_dma_init) |
| Function Description | Initialize the variables of ll_dma_init_t to default reset values. |
| Parameter | p_dma_init: pointer to structure variables to be reset |
| Return Value | None |
| Remarks | |

3.6 LL DUAL TIMER Generic Driver

3.6.1 DUAL TIMER Driver Structure

3.6.1.1 ll_dual_timer_init_t

The initialization structure ll_dual_timer_init_t of the LL DUAL TIMER driver is defined below:

Table 3-21 ll_dual_timer_init_t structure

| Data Field | Field Description | Value |
|-----------------------|--|---|
| uint32_t prescaler | Prescaler value, which can also be set with ll_dual_timer_set_prescaler() | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • LL_DUAL_TIMER_PRESCALER_DIV0 (fclk) • LL_DUAL_TIMER_PRESCALER_DIV16 (fclk/16) • LL_DUAL_TIMER_PRESCALER_DIV256 (fclk/256) |
| uint32_t counter_size | Timer bit width, which can also be set with ll_dual_timer_set_counter_size() | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • LL_DUAL_TIMER_COUNTERSIZE_16 (16 bits) • LL_DUAL_TIMER_COUNTERSIZE_32 (32 bits) |

| Data Field | Field Description | Value |
|-----------------------|---|---|
| uint32_t counter_mode | Counting mode, which can also be set with <code>ll_dual_timer_set_counter_mode()</code> | This parameter can be one of the following values: <ul style="list-style-type: none"> • <code>LL_DUAL_TIMER_FREERUNNING_MODE</code> (free counting mode) • <code>LL_DUAL_TIMER_PERIODIC_MODE</code> (period mode) |
| uint32_t auto_reload | Initial counting value, which can also be set with <code>ll_dual_timer_set_auto_reload()</code> | 0x0000_0000 to 0xFFFF_FFFF |

3.6.2 DUAL TIMER Driver APIs

The DUAL TIMER driver APIs are listed in the table below:

Table 3-22 DUAL TIMER driver APIs

| API Type | API Name | Description |
|---------------------------------|--|---|
| Initialization/Deinitialization | <code>ll_dual_timer_init()</code> | Initialize DUAL TIMER peripheral. |
| | <code>ll_dual_timer_deinit()</code> | Deinitialize DUAL TIMER peripheral to default. |
| | <code>ll_dual_timer_struct_init()</code> | Initialize the structure <code>dual_timer_init</code> to default. |

The sections below elaborate on these APIs.

3.6.2.1 ll_dual_timer_init

Table 3-23 ll_dual_timer_init API

| | |
|-----------------------------|---|
| Function Prototype | <code>error_status_t ll_dual_timer_init(dual_timer_regs_t *DUAL_TIMERx, ll_dual_timer_init_t *p_dual_timer_init)</code> |
| Function Description | Initialize DUAL TIMER peripheral according to specified parameters in <code>ll_dual_timer_init_t</code> . |
| Parameter | <p><code>DUAL_TIMERx</code>: DUAL TIMER peripheral instance</p> <p><code>p_dual_timer_init</code>: pointer to the variables of <code>ll_dual_timer_init_t</code>. The variable contains the configuration information of a specified DUAL TIMER.</p> |
| Return Value | <p><code>error_status_t</code> shows the enumeration type, which can be:</p> <ul style="list-style-type: none"> • <code>SUCCESS</code>: Initialization of DUAL TIMER peripheral registers succeeds. • <code>ERROR</code>: Initialization fails. |
| Remarks | |

3.6.2.2 ll_dual_timer_deinit

Table 3-24 ll_dual_timer_deinit API

| | |
|-----------------------------|--|
| Function Prototype | <code>error_status_t ll_dual_timer_deinit(dual_timer_regs_t *DUAL_TIMERx)</code> |
| Function Description | Deinitialize the DUAL TIMERx peripheral registers to default reset values. |

| | |
|---------------------|---|
| Parameter | DUAL_TIMERx: DUAL TIMER peripheral instance |
| Return Value | error_status_t shows the enumeration type, which can be: <ul style="list-style-type: none"> • SUCCESS: Deinitialization of DUAL TIMER peripheral registers succeeds. • ERROR: Deinitialization fails. |
| Remarks | |

3.6.2.3 ll_dual_timer_struct_init

Table 3-25 ll_dual_timer_struct_init API

| | |
|-----------------------------|---|
| Function Prototype | void ll_dual_timer_struct_init(ll_dual_timer_init_t *p_dual_timer_init) |
| Function Description | Initialize variables of ll_dual_timer_init_t to default reset values. |
| Parameter | p_dual_timer_init: pointer to structure variables to be reset |
| Return Value | None |
| Remarks | |

3.7 LL I2C Generic Driver

3.7.1 I2C Driver Structures

3.7.1.1 ll_i2c_init_t

The initialization structure ll_i2c_init_t of the LL I2C driver is defined below:

Table 3-26 ll_i2c_init_t structure

| Data Field | Field Description | Value |
|------------------------|---|---|
| uint32_t speed | Transfer speed, which can also be set with ll_i2c_configure_speed() | This parameter can be one of the following values: <ul style="list-style-type: none"> • LL_I2C_SPEED_100K (100 Kbps) • LL_I2C_SPEED_400K (400 Kbps) • LL_I2C_SPEED_1000K (1.0 Mbps) • LL_I2C_SPEED_2000K (2.0 Mbps) |
| uint32_t own_address | Local device address (slave mode), which can also be set with ll_i2c_set_own_address() | 7-bit address: 0x08 to 0x77 10-bit address: 0x008 to 0x077, 0x080 to 0x3FE |
| uint32_t own_addr_size | Local device address format (slave mode), which also can be set with ll_i2c_set_own_address() | This parameter can be one of the following values: <ul style="list-style-type: none"> • LL_I2C_OWADDRESS_7BIT (7-bit address) • LL_I2C_OWADDRESS_10BIT (10-bit address) |

3.7.2 I2C Driver APIs

The I2C driver APIs are listed in the table below:

Table 3-27 I2C driver APIs

| API Type | API Name | Description |
|---------------------------------|----------------------|---|
| Initialization/Deinitialization | ll_i2c_init() | Initialize I2C peripheral. |
| | ll_i2c_deinit() | Deinitialize I2C peripheral to default. |
| | ll_i2c_struct_init() | Initialize the structure i2c_init to default. |

The sections below elaborate on these APIs.

3.7.2.1 ll_i2c_init

Table 3-28 ll_i2c_init API

| | |
|-----------------------------|--|
| Function Prototype | error_status_t ll_i2c_init(i2c_regs_t *I2Cx, ll_i2c_init_t *p_i2c_init) |
| Function Description | Initialize I2C peripheral according to specified parameters in ll_i2c_init_t . |
| Parameter | I2Cx: I2C peripheral instance p_i2c_init: pointer to the variables of ll_i2c_init_t . The variable contains the configuration information of a specified I2C peripheral instance. |
| Return Value | error_status_t shows the enumeration type, which can be: <ul style="list-style-type: none"> • SUCCESS: Initialization of I2C peripheral registers succeeds. • ERROR: Initialization fails. |
| Remarks | |

3.7.2.2 ll_i2c_deinit

Table 3-29 ll_i2c_deinit API

| | |
|-----------------------------|--|
| Function Prototype | error_status_t ll_i2c_deinit(i2c_regs_t *I2Cx) |
| Function Description | Deinitialize the I2C peripheral registers to default reset values. |
| Parameter | I2Cx: I2C peripheral instance |
| Return Value | error_status_t shows the enumeration type, which can be: <ul style="list-style-type: none"> • SUCCESS: Deinitialization of I2C peripheral registers succeeds. • ERROR: Deinitialization fails. |
| Remarks | |

3.7.2.3 ll_i2c_struct_init

Table 3-30 ll_i2c_struct_init API

| | |
|---------------------------|--|
| Function Prototype | void ll_i2c_struct_init(ll_i2c_init_t *p_i2c_init) |
|---------------------------|--|

| | |
|-----------------------------|---|
| Function Description | Initialize variables of <code>ll_i2c_init_t</code> to default reset values. |
| Parameter | <code>p_i2c_init</code> : pointer to structure variables to be reset |
| Return Value | None |
| Remarks | |

3.8 LL MSIO Generic Driver

3.8.1 MSIO Driver Structure

3.8.1.1 ll_msio_init_t

The initialization structure `ll_msio_init_t` of the LL MSIO driver is defined below:

Table 3-31 ll_msio_init_t structure

| Data Field | Field Description | Value |
|---------------------------------|---|---|
| <code>uint32_t pin</code> | MSIO pins to be configured | <p>This parameter can be any combination of the following values:</p> <ul style="list-style-type: none"> • <code>LL_MSIO_PIN_0</code> (Pin 0) • <code>LL_MSIO_PIN_1</code> (Pin 1) • <code>LL_MSIO_PIN_2</code> (Pin 2) • <code>LL_MSIO_PIN_3</code> (Pin 3) • <code>LL_MSIO_PIN_4</code> (Pin 4) • <code>LL_MSIO_PIN_ALL</code> (all pins) |
| <code>uint32_t direction</code> | Direction of the selected pin (input/output), which can also be set with <code>ll_msio_set_pin_direction()</code> | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • <code>LL_MSIO_DIRECTION_NONE</code> (disable input/output) • <code>LL_MSIO_DIRECTION_INPUT</code> (enable input) • <code>LL_MSIO_DIRECTION_OUTPUT</code> (enable output) • <code>LL_MSIO_DIRECTION_INOUT</code> (enable input and output) |
| <code>uint32_t mode</code> | The designated operating mode of the selected pin, which can also be set with <code>ll_msio_set_pin_mode()</code> | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • <code>LL_MSIO_MODE_ANALOG</code> (analog mode) • <code>LL_MSIO_MODE_DIGITAL</code> (digital mode) |
| <code>uint32_t pull</code> | The type of the selected pull-up/pull-down resistor, which can also be set with <code>ll_msio_set_pin_pull()</code> | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • <code>LL_MSIO_PULL_NO</code> (no pull-up/pull-down resistor activated) • <code>LL_MSIO_PULL_UP</code> (activate pull-up resistor) • <code>LL_MSIO_PULL_DOWN</code> (activate pull-down resistor) |

3.8.2 MSIO Driver APIs

The MSIO driver APIs are listed in the table below:

Table 3-32 MSIO driver APIs

| API Type | API Name | Description |
|---------------------------------|-----------------------|--|
| Initialization/Deinitialization | ll_msio_init() | Initialize MSIO peripheral. |
| | ll_msio_deinit() | Deinitialize MSIO peripheral to default. |
| | ll_msio_struct_init() | Initialize the structure msio_init to default. |

The sections below elaborate on these APIs.

3.8.2.1 ll_msio_init

Table 3-33 ll_msio_init API

| | |
|-----------------------------|---|
| Function Prototype | error_status_t ll_msio_init(ll_msio_init_t *p_msio_init) |
| Function Description | Initialize MSIO peripheral according to specified parameters in ll_msio_init_t . |
| Parameter | p_msio_init: pointer to the variables of ll_msio_init_t . The variable contains the configuration information of a specified MSIO pin. |
| Return Value | error_status_t shows the enumeration type, which can be: <ul style="list-style-type: none"> • SUCCESS: Initialization of MSIO peripheral registers succeeds. • ERROR: Initialization fails. |
| Remarks | |

3.8.2.2 ll_msio_deinit

Table 3-34 ll_msio_deinit API

| | |
|-----------------------------|---|
| Function Prototype | error_status_t ll_msio_deinit(void) |
| Function Description | Deinitialize the MSIO peripheral registers to default reset values. |
| Parameter | None |
| Return Value | error_status_t shows the enumeration type, which can be: <ul style="list-style-type: none"> • SUCCESS: Deinitialization of MSIO peripheral registers succeeds. • ERROR: Deinitialization fails. |
| Remarks | |

3.8.2.3 ll_msio_struct_init

Table 3-35 ll_msio_struct_init API

| | |
|---------------------------|---|
| Function Prototype | void ll_msio_struct_init(ll_msio_init_t *p_msio_init) |
|---------------------------|---|

| | |
|-----------------------------|--|
| Function Description | Initialize variables of <code>ll_msio_init_t</code> to default reset values. |
| Parameter | <code>p_msio_init</code> : pointer to structure variables to be reset |
| Return Value | None |
| Remarks | |

3.9 LL AES Generic Driver

3.9.1 AES Driver Structure

3.9.1.1 ll_aes_init_t

The AES structure `ll_aes_init_t` is defined below:

Table 3-36 ll_aes_init_t structure

| Data Field | Field Description | Value |
|--------------------------------------|--|--|
| <code>uint32_t key_size</code> | Length of AES keys | This parameter can be one of the following values: <ul style="list-style-type: none"> • <code>LL_AES_KEY_SIZE_128</code> (128 bits) • <code>LL_AES_KEY_SIZE_192</code> (192 bits) • <code>LL_AES_KEY_SIZE_256</code> (256 bits) |
| <code>uint32_t *p_key</code> | AES key | This parameter is specified by developers. |
| <code>uint32_t *p_init_vector</code> | Initialization vector, valid in CBC mode | This parameter is specified by developers. |
| <code>uint32_t *p_key</code> | Random number seed | This parameter is specified by developers. |

3.9.2 AES Driver APIs

The AES driver APIs are listed in the table below:

Table 3-37 AES driver APIs

| API Type | API Name | Description |
|-------------------------------------|--|--|
| Initialization/ Deinitialization | <code>ll_aes_init()</code> | Initialize AES peripheral. |
| | <code>ll_aes_deinit()</code> | Deinitialize AES peripheral to default. |
| | <code>ll_aes_struct_init(ll_aes_init_t *aes_init)</code> | Initialize the structure <code>aes_init</code> to default. |

The sections below elaborate on these APIs.

3.9.2.1 ll_aes_init

Table 3-38 ll_aes_init API

| | |
|---------------------------|--|
| Function Prototype | <code>error_status_t ll_aes_init(aes_regs_t *AESx, ll_aes_init_t *p_aes_init)</code> |
|---------------------------|--|

| | |
|-----------------------------|--|
| Function Description | Initialize AES peripheral, and enable the peripheral and little-endian mode. |
| Parameter | AESx: AES peripheral instance |
| Return Value | error_status_t shows the enumeration type, which can be: <ul style="list-style-type: none"> • SUCCESS: Initialization of AES peripheral registers succeeds. • ERROR: Initialization fails. |
| Remarks | |

3.9.2.2 ll_aes_deinit

Table 3-39 ll_aes_deinit API

| | |
|-----------------------------|--|
| Function Prototype | error_status_t ll_aes_deinit(aes_regs_t *AESx) |
| Function Description | Deinitialize AES peripheral registers and set the register values as default. |
| Parameter | AESx: AES peripheral instance |
| Return Value | error_status_t shows the enumeration type, which can be: <ul style="list-style-type: none"> • SUCCESS: Deinitialization of AES peripheral registers succeeds. • ERROR: Deinitialization fails. |
| Remarks | |

3.9.2.3 ll_aes_struct_init

Table 3-40 ll_aes_struct_init API

| | |
|-----------------------------|--|
| Function Prototype | void ll_aes_struct_init(ll_aes_init_t *p_aes_init) |
| Function Description | Initialize variables of ll_aes_init_t to default values. |
| Parameter | p_aes_init: pointer to structure variables to be reset |
| Return Value | None |
| Remarks | |

3.10 LL PKC Generic Driver

3.10.1 PKC Driver Structures

3.10.1.1 ll_ecc_point_t

The ECC point coordinate structure ll_ecc_point_t is defined below:

Table 3-41 ll_ecc_point_t structure

| Data Field | Field Description | Value |
|----------------------------|-------------------------|--|
| uint32_t X[ECC_U32_LENGTH] | X position of ECC point | This parameter is specified by developers. |

| Data Field | Field Description | Value |
|----------------------------|-------------------------|--|
| uint32_t Y[ECC_U32_LENGTH] | Y position of ECC point | This parameter is specified by developers. |

3.10.1.2 ll_ecc_curve_init_t

The ECC structure ll_ecc_curve_init_t is defined below:

Table 3-42 ll_ecc_curve_init_t structure

| Data Field | Field Description | Value |
|-------------------------------------|--|--|
| uint32_t A[ECC_U32_LENGTH] | Operand A array (defines an elliptic curve with operand B array) | This parameter is specified by developers. |
| uint32_t B[ECC_U32_LENGTH] | Operand B array (defines an elliptic curve with operand A array) | This parameter is specified by developers. |
| uint32_t P[ECC_U32_LENGTH] | Prime number P | This parameter is specified by developers. |
| uin32_t PRSquare[ECC_U32_LENGTH] | Modulo operation of P | This parameter is specified by developers. |
| uin32_t ConstP | Constant P in Montgomery multiplication | This parameter is specified by developers. |
| uint32_t N[ECC_U32_LENGTH] | Prime number N | This parameter is specified by developers. |
| uin32_t NRSquare[ECC_U32_LENGTH] | Modulo operation of N | This parameter is specified by developers. |
| uin32_t ConstN | Constant N in Montgomery multiplication | This parameter is specified by developers. |
| uint32_t H | Factor H | This parameter is specified by developers. |
| ll_ecc_point_t G | Base point of an elliptic curve | This parameter is specified by developers. |

3.10.1.3 ll_pkc_init_t

The initialization structure ll_pkc_init_t of the LL PKC driver is defined below:

Table 3-43 ll_pkc_init_t structure

| Data Field | Field Description | Value |
|-----------------------------------|---|--|
| ll_ecc_curve_init_t *ecc_curve | Parameters of an elliptic curve to be configured. See " Section 3.10.1.2 ll_ecc_curve_init_t " for reference. | This parameter is specified by developers. |
| uint32_t data_bits | Data bit width | 256 to 2048 |

3.10.2 PKC Driver APIs

The PKC driver APIs are listed in the table below:

Table 3-44 PKC driver APIs

| API Type | API Name | Description |
|---------------------------------|---|---|
| Initialization/Deinitialization | ll_pkc_init() | Initialize PKC peripheral. |
| | ll_pkc_deinit() | Deinitialize PKC peripheral to default. |
| | ll_pkc_struct_init(ll_pkc_init_t *pkc_init) | Initialize the structure pkc_init to default. |

The sections below elaborate on these APIs.

3.10.2.1 ll_pkc_init

Table 3-45 ll_pkc_init API

| | |
|-----------------------------|--|
| Function Prototype | error_status_t ll_pkc_init(pkc_regs_t *PKCx, ll_pkc_init_t *p_pkc_init) |
| Function Description | Initialize PKC peripheral, and enable the peripheral and little-endian mode. |
| Parameter | PKCx: PKC peripheral instance |
| Return Value | error_status_t shows the enumeration type, which can be: <ul style="list-style-type: none"> • SUCCESS: Initialization of PKC peripheral registers succeeds. • ERROR: Initialization fails. |
| Remarks | |

3.10.2.2 ll_pkc_deinit

Table 3-46 ll_pkc_deinit API

| | |
|-----------------------------|--|
| Function Prototype | error_status_t ll_pkc_deinit(pkc_regs_t *PKCx) |
| Function Description | Deinitialize the PKC peripheral registers to default reset values. |
| Parameter | PKCx: PKC peripheral instance |
| Return Value | error_status_t shows the enumeration type, which can be: <ul style="list-style-type: none"> • SUCCESS: Deinitialization of PKC peripheral registers succeeds. • ERROR: Deinitialization fails. |
| Remarks | |

3.10.2.3 ll_pkc_struct_init

Table 3-47 ll_pkc_struct_init API

| | |
|-----------------------------|--|
| Function Prototype | void ll_pkc_struct_init(ll_pkc_init_t *p_pkc_init) |
| Function Description | Initialize variables of ll_pkc_init_t to default reset values. |
| Parameter | p_pkc_init: pointer to structure variables to be reset |

| | |
|--------------|------|
| Return Value | None |
| Remarks | |

3.11 LL PWM Generic Driver

3.11.1 PWM Driver Structures

3.11.1.1 ll_pwm_channel_init_t

The initialization structure ll_pwm_channel_init_t of the LL PWM driver is defined below:

Table 3-48 ll_pwm_channel_init_t structure

| Data Field | Field Description | Value |
|-------------------------|--|---|
| uint32_t duty | Duty ratio, which can also be set with APIs such as ll_pwm_set_compare_a0() | 0 to 100 |
| uint32_t drive_polarity | Drive polarity, which can also be set with APIs such as ll_pwm_enable_positive_drive_channel_a() | This parameter can be one of the following values: <ul style="list-style-type: none"> • LL_PWM_DRIVEPOLARITY_NEGATIVE (negative drive polarity) • LL_PWM_DRIVEPOLARITY_POSITIVE (positive drive polarity) |

3.11.1.2 ll_pwm_init_t

The initialization structure ll_pwm_init_t of the LL PWM driver is defined below:

Table 3-49 ll_pwm_init_t structure

| Data Field | Field Description | Value |
|---------------------|---|--|
| uint32_t mode | Output mode, which can also be set with ll_pwm_set_mod() | This parameter can be one of the following values: <ul style="list-style-type: none"> • LL_PWM_FLICKER_MODE (normal mode) • LL_PWM_BREATH_MODE (breathing mode) |
| uint32_t align | Alignment mode (to be set by developers during initialization) | This parameter can be one of the following values: <ul style="list-style-type: none"> • LL_PWM_EDGE_ALIGNED (left-edge-aligned) • LL_PWM_CENTER_ALIGNED (center-aligned) |
| uint32_t prescaler | Output period, which can also be set with ll_pwm_set_prescaler() | 0x0000_0000 to 0xFFFF_FFFF. A parameter on a multiple of 128 is suggested. |
| uint32_t bprescaler | Breathing period (the time required during the duty ratio increasing from 0 to 100), which can also be set with ll_pwm_set_breath_prescaler() | 0x0000_0000 to 0xFFFF_FFFF. A parameter on a multiple of the period x 128 is suggested. |

| Data Field | Field Description | Value |
|---------------------------------|--|---|
| uint32_t hprescaler | Breathing hold period (the hold time between two breathing periods), which can also be set with <code>ll_pwm_set_hold_prescaler()</code> | 0x0000_0000 to 0xFFFF_FFFF. A parameter on a multiple of the period is suggested. |
| ll_pwm_channel_init_t channel_a | Initialization structure of channel A | See " Section 3.11.1.1 ll_pwm_channel_init_t ". |
| ll_pwm_channel_init_t channel_b | Initialization structure of channel B | See " Section 3.11.1.1 ll_pwm_channel_init_t ". |
| ll_pwm_channel_init_t channel_c | Initialization structure of channel C | See " Section 3.11.1.1 ll_pwm_channel_init_t ". |

3.11.2 PWM Driver APIs

The PWM driver APIs are listed in the table below:

Table 3-50 PWM driver APIs

| API Type | API Name | Description |
|---------------------------------|-----------------------------------|--|
| Initialization/Deinitialization | <code>ll_pwm_init()</code> | Initialize PWM peripheral. |
| | <code>ll_pwm_deinit()</code> | Deinitialize PWM peripheral to default. |
| | <code>ll_pwm_struct_init()</code> | Initialize variables of <code>ll_pwm_channel_init_t</code> to default. |

The sections below elaborate on these APIs.

3.11.2.1 ll_pwm_init

Table 3-51 ll_pwm_init API

| | |
|-----------------------------|--|
| Function Prototype | <code>error_status_t ll_pwm_init(pwm_regs_t *PWMx, ll_pwm_init_t *p_pwm_init)</code> |
| Function Description | Initialize PWM peripheral according to specified parameters in <code>ll_pwm_init_t</code> . |
| Parameter | <p>PWMx: PWM peripheral instance</p> <p>p_pwm_init: pointer to the variables of <code>ll_pwm_init_t</code>. The variable contains the configuration information of a specified PWM peripheral instance.</p> |
| Return Value | <p><code>error_status_t</code> shows the enumeration type, which can be:</p> <ul style="list-style-type: none"> • SUCCESS: Initialization of PWM peripheral registers succeeds. • ERROR: Initialization fails. |
| Remarks | |

3.11.2.2 ll_pwm_deinit

Table 3-52 ll_pwm_deinit API

| | |
|-----------------------------|--|
| Function Prototype | error_status_t ll_pwm_deinit(pwm_regs_t *PWMx) |
| Function Description | Deinitialize the PWM peripheral registers to default reset values. |
| Parameter | PWMx: PWM peripheral instance |
| Return Value | error_status_t shows the enumeration type, which can be: <ul style="list-style-type: none"> • SUCCESS: Deinitialization of PWM peripheral registers succeeds. • ERROR: Deinitialization fails. |
| Remarks | |

3.11.2.3 ll_pwm_struct_init

Table 3-53 ll_pwm_struct_init API

| | |
|-----------------------------|--|
| Function Prototype | void ll_pwm_struct_init(ll_pwm_init_t *p_pwm_init) |
| Function Description | Initialize variables of ll_pwm_init_t to default reset values. |
| Parameter | p_pwm_init: pointer to structure variables to be reset |
| Return Value | None |
| Remarks | |

3.12 LL SPI Generic Driver

3.12.1 SPI Driver Structures

3.12.1.1 ll_spim_init_t

The initialization structure ll_spim_init_t of the LL SPIM driver is defined below:

Table 3-54 ll_spim_init_t structure

| Data Field | Field Description | Value |
|-----------------------------|---|--|
| uint32_t transfer_direction | Data transfer direction, which can also be set with ll_spi_set_transfer_direction() | This parameter can be one of the following values: <ul style="list-style-type: none"> • LL_SSI_FULL_DUPLEX (full duplex) • LL_SSI_SIMPLEX_TX (simplex TX) • LL_SSI_SIMPLEX_RX (simplex RX) • LL_SSI_READ_EEPROM (reading EEPROM) |
| uint32_t data_size | Data TX bit width, which can also be set with ll_spi_set_data_size() | This parameter can be one of the following values: <ul style="list-style-type: none"> • LL_SSI_DATASIZE_4BIT (4 bits) • LL_SSI_DATASIZE_5BIT (5 bits) • LL_SSI_DATASIZE_6BIT (6 bits) |

| Data Field | Field Description | Value |
|-------------------------|---|--|
| | | <ul style="list-style-type: none"> • LL_SSI_DATASIZE_7BIT (7 bits) • LL_SSI_DATASIZE_8BIT (8 bits) • LL_SSI_DATASIZE_9BIT (9 bits) • LL_SSI_DATASIZE_10BIT (10 bits) • LL_SSI_DATASIZE_11BIT (11 bits) • LL_SSI_DATASIZE_12BIT (12 bits) • LL_SSI_DATASIZE_13BIT (13 bits) • LL_SSI_DATASIZE_14BIT (14 bits) • LL_SSI_DATASIZE_15BIT (15 bits) • LL_SSI_DATASIZE_16BIT (16 bits) • LL_SSI_DATASIZE_17BIT (17 bits) • LL_SSI_DATASIZE_18BIT (18 bits) • LL_SSI_DATASIZE_19BIT (19 bits) • LL_SSI_DATASIZE_20BIT (20 bits) • LL_SSI_DATASIZE_21BIT (21 bits) • LL_SSI_DATASIZE_22BIT (22 bits) • LL_SSI_DATASIZE_23BIT (23 bits) • LL_SSI_DATASIZE_24BIT (24 bits) • LL_SSI_DATASIZE_25BIT (25 bits) • LL_SSI_DATASIZE_26BIT (26 bits) • LL_SSI_DATASIZE_27BIT (27 bits) • LL_SSI_DATASIZE_28BIT (28 bits) • LL_SSI_DATASIZE_29BIT (29 bits) • LL_SSI_DATASIZE_30BIT (30 bits) • LL_SSI_DATASIZE_31BIT (31 bits) • LL_SSI_DATASIZE_32BIT (32 bits) |
| uint32_t clock_polarity | Clock polarity, which can also be set with <code>ll_spi_set_clock_polarity()</code> | This parameter can be one of the following values: <ul style="list-style-type: none"> • LL_SSI_SCPOL_LOW (clock idle at a low level) • LL_SSI_SCPOL_HIGH (clock idle at a high level) |
| uint32_t clock_phase | Clock phase, which can also be set with <code>ll_spi_set_clock_phase()</code> | This parameter can be one of the following values: <ul style="list-style-type: none"> • LL_SSI_SCPHA_1EDGE (data capture edge at the first clock transition) • LL_SSI_SCPHA_2EDGE (data capture edge at the second clock transition) |

| Data Field | Field Description | Value |
|-----------------------|---|---|
| uint32_t slave_select | The selected slave, which can also be set with <code>ll_spi_enable_ss()</code> | This parameter can be one of the following values: <ul style="list-style-type: none"> • LL_SSI_SLAVE0 (Slave 0) • LL_SSI_SLAVE1 (Slave 1) |
| uint32_t baud_rate | Baud rate prescaler, which can also be set with <code>ll_spi_set_baud_rate_prescaler()</code> | Even numbers from 2 to 65534 |

3.12.1.2 ll_spis_init_t

The initialization structure `ll_spis_init_t` of the LL SPIS driver is defined below:

Table 3-55 ll_spis_init_t structure

| Data Field | Field Description | Value |
|--------------------|---|---|
| uint32_t data_size | Data TX bit width, which can also be set with <code>ll_spi_set_data_size()</code> | This parameter can be one of the following values: <ul style="list-style-type: none"> • LL_SSI_DATASIZE_4BIT (4 bits) • LL_SSI_DATASIZE_5BIT (5 bits) • LL_SSI_DATASIZE_6BIT (6 bits) • LL_SSI_DATASIZE_7BIT (7 bits) • LL_SSI_DATASIZE_8BIT (8 bits) • LL_SSI_DATASIZE_9BIT (9 bits) • LL_SSI_DATASIZE_10BIT (10 bits) • LL_SSI_DATASIZE_11BIT (11 bits) • LL_SSI_DATASIZE_12BIT (12 bits) • LL_SSI_DATASIZE_13BIT (13 bits) • LL_SSI_DATASIZE_14BIT (14 bits) • LL_SSI_DATASIZE_15BIT (15 bits) • LL_SSI_DATASIZE_16BIT (16 bits) • LL_SSI_DATASIZE_17BIT (17 bits) • LL_SSI_DATASIZE_18BIT (18 bits) • LL_SSI_DATASIZE_19BIT (19 bits) • LL_SSI_DATASIZE_20BIT (20 bits) • LL_SSI_DATASIZE_21BIT (21 bits) • LL_SSI_DATASIZE_22BIT (22 bits) • LL_SSI_DATASIZE_23BIT (23 bits) • LL_SSI_DATASIZE_24BIT (24 bits) • LL_SSI_DATASIZE_25BIT (25 bits) |

| Data Field | Field Description | Value |
|-------------------------|---|---|
| | | <ul style="list-style-type: none"> • LL_SSI_DATASIZE_26BIT (26 bits) • LL_SSI_DATASIZE_27BIT (27 bits) • LL_SSI_DATASIZE_28BIT (28 bits) • LL_SSI_DATASIZE_29BIT (29 bits) • LL_SSI_DATASIZE_30BIT (30 bits) • LL_SSI_DATASIZE_31BIT (31 bits) • LL_SSI_DATASIZE_32BIT (32 bits) |
| uint32_t clock_polarity | Clock polarity, which can also be set with <code>ll_spi_set_clock_polarity()</code> | This parameter can be one of the following values: <ul style="list-style-type: none"> • LL_SSI_SCPOL_LOW (clock idle at a low level) • LL_SSI_SCPOL_HIGH (clock idle at a high level) |
| uint32_t clock_phase | Clock phase, which can also be set with <code>ll_spi_set_clock_phase()</code> | This parameter can be one of the following values: <ul style="list-style-type: none"> • LL_SSI_SCPHA_1EDGE (data capture edge at the first clock transition) • LL_SSI_SCPHA_2EDGE (data capture edge at the second clock transition) |

3.12.1.3 ll_qspi_init_t

The initialization structure `ll_qspi_init_t` for LL QSPI peripheral is defined below:

Table 3-56 ll_qspi_init_t structure

| Data Field | Field Description | Value |
|-----------------------------|--|--|
| uint32_t transfer_direction | Data transfer direction, which can also be set with <code>ll_spi_set_transfer_direction()</code> | This parameter can be one of the following values: <ul style="list-style-type: none"> • LL_SSI_FULL_DUPLEX (full duplex) • LL_SSI_SIMPLEX_TX (simplex TX) • LL_SSI_SIMPLEX_RX (simplex RX) • LL_SSI_READ_EEPROM (reading EEPROM) |
| uint32_t instruction_size | Instruction bit width, which can also be set with <code>ll_spi_set_instruction_size()</code> | This parameter can be one of the following values: <ul style="list-style-type: none"> • LL_SSI_INSTSIZE_0BIT (0 bit) • LL_SSI_INSTSIZE_4BIT (4 bits) • LL_SSI_INSTSIZE_8BIT (8 bits) • LL_SSI_INSTSIZE_16BIT (16 bits) |
| uint32_t address_size | Address bit width, which can also be set with <code>ll_spi_set_address_size()</code> | This parameter can be one of the following values: <ul style="list-style-type: none"> • LL_SSI_ADDRSIZE_0BIT (0 bit) • LL_SSI_ADDRSIZE_4BIT (4 bits) |

| Data Field | Field Description | Value |
|---------------------------------------|---|--|
| | | <ul style="list-style-type: none"> • LL_SSI_ADDR_SIZE_8BIT (8 bits) • LL_SSI_ADDR_SIZE_12BIT (12 bits) • LL_SSI_ADDR_SIZE_16BIT (16 bits) • LL_SSI_ADDR_SIZE_20BIT (20 bits) • LL_SSI_ADDR_SIZE_24BIT (24 bits) • LL_SSI_ADDR_SIZE_28BIT (28 bits) • LL_SSI_ADDR_SIZE_32BIT (32 bits) |
| uint32_t inst_addr_transfer_format | Transfer format for instructions and addresses, which can also be set with ll_spi_set_inst_transfer_format() | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • LL_SSI_INST_ADDR_ALL_IN_SPI (instructions and addresses transferred through SPI) • LL_SSI_INST_IN_SPI_ADDR_IN_SPIFRF (instructions transferred through SPI, and addresses through Dual/Quad SPI) • LL_SSI_INST_ADDR_ALL_IN_SPIFRF (instructions and addresses transferred through Dual/Quad SPI) |
| uint32_t wait_cycles | Clock waiting cycles (active for RX through Dual/Quad SPI in simplex mode), which can also be set with ll_spi_set_wait_cycles() | 0 to 31 |
| uint32_t data_size | Data TX bit width, which can also be set with ll_spi_set_data_size() | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • LL_SSI_DATA_SIZE_4BIT (4 bits) • LL_SSI_DATA_SIZE_5BIT (5 bits) • LL_SSI_DATA_SIZE_6BIT (6 bits) • LL_SSI_DATA_SIZE_7BIT (7 bits) • LL_SSI_DATA_SIZE_8BIT (8 bits) • LL_SSI_DATA_SIZE_9BIT (9 bits) • LL_SSI_DATA_SIZE_10BIT (10 bits) • LL_SSI_DATA_SIZE_11BIT (11 bits) • LL_SSI_DATA_SIZE_12BIT (12 bits) • LL_SSI_DATA_SIZE_13BIT (13 bits) • LL_SSI_DATA_SIZE_14BIT (14 bits) • LL_SSI_DATA_SIZE_15BIT (15 bits) • LL_SSI_DATA_SIZE_16BIT (16 bits) • LL_SSI_DATA_SIZE_17BIT (17 bits) • LL_SSI_DATA_SIZE_18BIT (18 bits) • LL_SSI_DATA_SIZE_19BIT (19 bits) |

| Data Field | Field Description | Value |
|--------------------------|---|---|
| | | <ul style="list-style-type: none"> • LL_SSI_DATASIZE_20BIT (20 bits) • LL_SSI_DATASIZE_21BIT (21 bits) • LL_SSI_DATASIZE_22BIT (22 bits) • LL_SSI_DATASIZE_23BIT (23 bits) • LL_SSI_DATASIZE_24BIT (24 bits) • LL_SSI_DATASIZE_25BIT (25 bits) • LL_SSI_DATASIZE_26BIT (26 bits) • LL_SSI_DATASIZE_27BIT (27 bits) • LL_SSI_DATASIZE_28BIT (28 bits) • LL_SSI_DATASIZE_29BIT (29 bits) • LL_SSI_DATASIZE_30BIT (30 bits) • LL_SSI_DATASIZE_31BIT (31 bits) • LL_SSI_DATASIZE_32BIT (32 bits) |
| uint32_t clock_polarity | Clock polarity, which can also be set with <code>ll_spi_set_clock_polarity()</code> | This parameter can be one of the following values: <ul style="list-style-type: none"> • LL_SSI_SCPOL_LOW (clock idle at a low level) • LL_SSI_SCPOL_HIGH (clock idle at a high level) |
| uint32_t clock_phase | Clock phase, which can also be set with <code>ll_spi_set_clock_phase()</code> | This parameter can be one of the following values: <ul style="list-style-type: none"> • LL_SSI_SCPHA_1EDGE (data capture edge at the first clock transition) • LL_SSI_SCPHA_2EDGE (data capture edge at the second clock transition) |
| uint32_t baud_rate | Baud rate prescaler, which can also be set with <code>ll_spi_set_baud_rate_prescaler()</code> | Even numbers from 2 to 65534 |
| uint32_t rx_sample_delay | RX delayed acquisition | 0x0 to 0x7 |

3.12.2 SPI Driver APIs

The SPI driver APIs are listed in the table below:

Table 3-57 SPI driver APIs

| API Type | API Name | Description |
|---------------------------------|------------------------------------|---|
| Initialization/Deinitialization | <code>ll_spim_init()</code> | Initialize SPIM peripheral. |
| | <code>ll_spim_deinit()</code> | Deinitialize SPIM peripheral to default. |
| | <code>ll_spim_struct_init()</code> | Initialize variables of <code>ll_spim_init_t</code> to default. |
| | <code>ll_spis_init()</code> | Initialize SPIS peripheral. |
| | <code>ll_spis_deinit()</code> | Deinitialize SPIS peripheral to default. |

| API Type | API Name | Description |
|----------|-----------------------|--|
| | ll_spis_struct_init() | Initialize variables of ll_spis_init_t to default. |
| | ll_qspi_init() | Initialize QSPI peripheral. |
| | ll_qspi_deinit() | Deinitialize QSPI peripheral to default. |
| | ll_qspi_struct_init() | Initialize variables of ll_qspi_init_t to default. |

The sections below elaborate on these APIs.

3.12.2.1 ll_spim_init

Table 3-58 ll_spim_init API

| | |
|-----------------------------|--|
| Function Prototype | error_status_t ll_spim_init(ssi_regs_t *SPiX, ll_spim_init_t *p_spi_init) |
| Function Description | Initialize SPIM peripheral according to specified parameters in ll_spim_init_t. |
| Parameter | SPiX: SPIM peripheral instance p_spi_init: pointer to the variables of ll_spim_init_t. The variable contains the configuration information of a specified SPI peripheral instance. |
| Return Value | error_status_t shows the enumeration type, which can be: <ul style="list-style-type: none"> • SUCCESS: Initialization of SPI peripheral registers succeeds. • ERROR: Initialization fails. |
| Remarks | |

3.12.2.2 ll_spim_deinit

Table 3-59 ll_spim_deinit API

| | |
|-----------------------------|--|
| Function Prototype | error_status_t ll_spim_deinit(ssi_regs_t *SPiX) |
| Function Description | Deinitialize the SPI peripheral registers to default reset values. |
| Parameter | SPiX: SPIM peripheral instance |
| Return Value | error_status_t shows the enumeration type, which can be: <ul style="list-style-type: none"> • SUCCESS: Deinitialization of SPI peripheral registers succeeds. • ERROR: Deinitialization fails. |
| Remarks | |

3.12.2.3 ll_spim_struct_init

Table 3-60 ll_spim_struct_init API

| | |
|-----------------------------|---|
| Function Prototype | void ll_spim_struct_init(ll_spim_init_t *p_spi_init) |
| Function Description | Initialize variables of ll_spim_init_t to default reset values. |

| | |
|---------------------|--|
| Parameter | p_spi_init: pointer to structure variables to be reset |
| Return Value | None |
| Remarks | |

3.12.2.4 ll_spis_init

Table 3-61 ll_spis_init API

| | |
|-----------------------------|--|
| Function Prototype | error_status_t ll_spis_init(ssi_regs_t *SPix, ll_spis_init_t *p_spi_init) |
| Function Description | Initialize SPIS peripheral according to specified parameters in ll_spis_init_t . |
| Parameter | SPix: SPIS peripheral instance p_spi_init: pointer to the variables of ll_spis_init_t . The variable contains the configuration information of a specified SPI peripheral instance. |
| Return Value | error_status_t shows the enumeration type, which can be: <ul style="list-style-type: none"> • SUCCESS: Initialization of SPI peripheral registers succeeds. • ERROR: Initialization fails. |
| Remarks | |

3.12.2.5 ll_spis_deinit

Table 3-62 ll_spis_deinit API

| | |
|-----------------------------|--|
| Function Prototype | error_status_t ll_spis_deinit(ssi_regs_t *SPix) |
| Function Description | Deinitialize the SPI peripheral registers to default reset values. |
| Parameter | SPix: SPIS peripheral instance |
| Return Value | error_status_t shows the enumeration type, which can be: <ul style="list-style-type: none"> • SUCCESS: Deinitialization of SPI peripheral registers succeeds. • ERROR: Deinitialization fails. |
| Remarks | |

3.12.2.6 ll_spis_struct_init

Table 3-63 ll_spis_struct_init API

| | |
|-----------------------------|---|
| Function Prototype | void ll_spis_struct_init(ll_spis_init_t *p_spi_init) |
| Function Description | Initialize variables of ll_spis_init_t to default reset values. |
| Parameter | p_spi_init: pointer to structure variables to be reset |
| Return Value | None |
| Remarks | |

3.12.2.7 ll_qspi_init

Table 3-64 ll_qspi_init API

| | |
|-----------------------------|--|
| Function Prototype | error_status_t ll_qspi_init(ssi_regs_t *SPiX, ll_qspi_init_t *p_spi_init) |
| Function Description | Initialize QSPI peripheral according to specified parameters in ll_qspi_init_t. |
| Parameter | SPiX: QSPI peripheral instance p_spi_init: pointer to the variables of ll_qspi_init_t. The variable contains the configuration information of a specified SPI peripheral instance. |
| Return Value | error_status_t shows the enumeration type, which can be: <ul style="list-style-type: none"> • SUCCESS: Initialization of SPI peripheral registers succeeds. • ERROR: Initialization fails. |
| Remarks | |

3.12.2.8 ll_qspi_deinit

Table 3-65 ll_qspi_deinit API

| | |
|-----------------------------|--|
| Function Prototype | error_status_t ll_qspi_deinit(ssi_regs_t *SPiX) |
| Function Description | Deinitialize the SPI peripheral registers to default reset values. |
| Parameter | SPiX: QSPI peripheral instance |
| Return Value | error_status_t shows the enumeration type, which can be: <ul style="list-style-type: none"> • SUCCESS: Deinitialization of SPI peripheral registers succeeds. • ERROR: Deinitialization fails. |
| Remarks | |

3.12.2.9 ll_qspi_struct_init

Table 3-66 ll_qspi_struct_init API

| | |
|-----------------------------|---|
| Function Prototype | void ll_qspi_struct_init(ll_qspi_init_t *p_spi_init) |
| Function Description | Initialize variables of ll_qspi_init_t to default reset values. |
| Parameter | p_spi_init: pointer to structure variables to be reset |
| Return Value | None |
| Remarks | |

3.13 LL TIMER Generic Driver

3.13.1 TIMER Driver Structure

3.13.1.1 ll_timer_init_t

The initialization structure ll_timer_init_t of the LL TIMER driver is defined below:

Table 3-67 ll_timer_init_t structure

| Data Field | Field Description | Value |
|----------------------|---|----------------------------|
| uint32_t auto_reload | Initial counting value, which can also be set with ll_tim_set_auto_reload() | 0x0000_0000 to 0xFFFF_FFFF |

3.13.2 TIMER Driver APIs

The TIMER driver APIs are listed in the table below:

Table 3-68 TIMER driver APIs

| API Type | API Name | Description |
|---------------------------------|------------------------|---|
| Initialization/Deinitialization | ll_timer_init() | Initialize TIMER peripherals. |
| | ll_timer_deinit() | Deinitialize TIMER peripheral to default. |
| | ll_timer_struct_init() | Initialize variables of ll_timer_init_t to default. |

The sections below elaborate on these APIs.

3.13.2.1 ll_timer_init

Table 3-69 ll_timer_init API

| | |
|-----------------------------|--|
| Function Prototype | error_status_t ll_timer_init(timer_regs_t *TIMERx, ll_timer_init_t *p_timer_init) |
| Function Description | Initialize TIMER peripheral according to specified parameters in ll_timer_init_t . |
| Parameter | TIMERx: TIMER peripheral instance p_timer_init: pointer to the variables of ll_timer_init_t . The variable contains the configuration information of a specified TIMER. |
| Return Value | error_status_t shows the enumeration type, which can be: <ul style="list-style-type: none"> • SUCCESS: Initialization of TIMER peripheral registers succeeds. • ERROR: Initialization fails. |
| Remarks | |

3.13.2.2 ll_timer_deinit

Table 3-70 ll_timer_deinit API

| | |
|-----------------------------|---|
| Function Prototype | error_status_t ll_timer_deinit(timer_regs_t *TIMERx) |
| Function Description | Deinitialize the TIMERx peripheral registers to default reset values. |
| Parameter | TIMERx: TIMER peripheral instance |

| | |
|---------------------|---|
| Return Value | <p>error_status_t shows the enumeration type, which can be:</p> <ul style="list-style-type: none"> • SUCCESS: Deinitialization of TIMER peripheral registers succeeds. • ERROR: Deinitialization fails. |
| Remarks | |

3.13.2.3 ll_timer_struct_init

Table 3-71 ll_timer_struct_init API

| | |
|-----------------------------|--|
| Function Prototype | void ll_timer_struct_init(ll_timer_init_t *p_timer_init) |
| Function Description | Initialize variables of ll_timer_init_t to default reset values. |
| Parameter | p_timer_init: pointer to structure variables to be reset |
| Return Value | None |
| Remarks | |

3.14 LL UART Generic Driver

3.14.1 UART Driver Structure

3.14.1.1 ll_uart_init_t

The initialization structure ll_uart_init_t of the LL UART driver is defined below:

Table 3-72 ll_uart_init_t structure

| Data Field | Field Description | Value |
|-----------------------|---|--|
| uint32_t baud_rate | Baud rate, which can also be set with ll_uart_set_baud_rate() | 9600 – 921600 |
| uint32_t data_bits | Data bit width, which can also be set with ll_uart_set_data_bits_length() | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • LL_UART_DATABITS_5B (5 bits) • LL_UART_DATABITS_6B (6 bits) • LL_UART_DATABITS_7B (7 bits) • LL_UART_DATABITS_8B (8 bits) |
| uint32_t stop_bits | Stop bit width, which can also be set with ll_uart_set_stop_bits_length() | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • LL_UART_STOPBITS_1 (1 bit) • LL_UART_STOPBITS_1_5 (1.5 bits) • LL_UART_STOPBITS_2 (2 bits) |
| uint32_t parity | Parity bit, which can also be set with ll_uart_set_parity() | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • LL_UART_PARITY_NONE (no parity) |

| Data Field | Field Description | Value |
|--------------------------|--|--|
| | | <ul style="list-style-type: none"> LL_UART_PARITY_ODD (odd parity) LL_UART_PARITY_EVEN (even parity) LL_UART_PARITY_SP0 (parity bit = 0) LL_UART_PARITY_SP1 (parity bit = 1) |
| uint32_t hw_flow_ctrl | Flow control, which can also be set with ll_uart_set_hw_flow_ctrl() | This parameter can be one of the following values: <ul style="list-style-type: none"> LL_UART_HWCONTROL_NONE (no flow control) LL_UART_HWCONTROL_RTS_CTS (automatic flow control) |

3.14.2 UART Driver APIs

The UART driver APIs are listed in the table below:

Table 3-73 UART driver APIs

| API Type | API Name | Description |
|---------------------------------|-----------------------|--|
| Initialization/Deinitialization | ll_uart_init() | Initialize UART peripherals. |
| | ll_uart_deinit() | Deinitialize UART peripheral to default. |
| | ll_uart_struct_init() | Initialize variables of ll_uart_init_t to default. |

The sections below elaborate on these APIs.

3.14.2.1 ll_uart_init

Table 3-74 ll_uart_init API

| | |
|-----------------------------|---|
| Function Prototype | error_status_t ll_uart_init(uart_regs_t *UARTx, ll_uart_init_t *p_uart_init) |
| Function Description | Initialize UART peripheral according to specified parameters in ll_uart_init_t . |
| Parameter | UARTx: UART peripheral instance p_uart_init: pointer to the variables of ll_uart_init_t . The variable contains the configuration information of a specified UART peripheral instance. |
| Return Value | error_status_t shows the enumeration type, which can be: <ul style="list-style-type: none"> SUCCESS: Initialization of UART peripheral registers succeeds. ERROR: Initialization fails. |
| Remarks | |

3.14.2.2 ll_uart_deinit

Table 3-75 ll_uart_deinit API

| | |
|-----------------------------|---|
| Function Prototype | error_status_t ll_uart_deinit(uart_regs_t *UARTx) |
| Function Description | Deinitialize the UART peripheral registers to default reset values. |

| | |
|---------------------|---|
| Parameter | UARTx: UART peripheral instance |
| Return Value | error_status_t shows the enumeration type, which can be: <ul style="list-style-type: none"> • SUCCESS: Deinitialization of UART peripheral registers succeeds. • ERROR: Deinitialization fails. |
| Remarks | |

3.14.2.3 ll_uart_struct_init

Table 3-76 ll_uart_struct_init API

| | |
|-----------------------------|---|
| Function Prototype | void ll_uart_struct_init(ll_uart_init_t *p_uart_init) |
| Function Description | Initialize variables of ll_uart_init_t to default reset values. |
| Parameter | p_uart_init: pointer to structure variables to be reset |
| Return Value | None |
| Remarks | |

3.15 LL I2S Generic Driver

3.15.1 I2S Driver Structure

3.15.1.1 ll_i2s_init_t

The initialization structure ll_i2c_init_t of the LL I2S driver is defined below:

Table 3-77 ll_i2s_init_t structure

| Data Field | Field Description | Value |
|----------------------|-------------------|---|
| uint32_t rxdata_size | Length of RX data | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • LL_I2S_DATASIZE_IGNORE • LL_I2S_DATASIZE_12BIT • LL_I2S_DATASIZE_16BIT • LL_I2S_DATASIZE_20BIT • LL_I2S_DATASIZE_24BIT • LL_I2S_DATASIZE_32BIT <p>Note:</p> <ul style="list-style-type: none"> • When data_size = I2S_DATASIZE_12BIT (12 bits), the transmitted data is 16-bit aligned and stored, with the higher 4-bit data ignored. The WSS of hardware is 16 SCLK cycles, with the higher 4-bit ignored. |

| Data Field | Field Description | Value |
|-----------------------|-------------------|---|
| | | <ul style="list-style-type: none"> When data_size = I2S_DATASIZE_20BIT (20 bits), the transmitted data is 32-bit aligned and stored, with the higher 12-bit data ignored. The WSS of hardware is 24 SCLK cycles, with the higher 4-bit ignored. <p>When data_size = I2S_DATASIZE_24BIT (24 bits), the transmitted data is 32-bit aligned and stored, with the higher 8-bit data ignored. The WSS of hardware is 24 SCLK cycles.</p> |
| uint32_t txdata_size | Length of TX data | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> LL_I2S_DATASIZE_IGNORE LL_I2S_DATASIZE_12BIT LL_I2S_DATASIZE_16BIT LL_I2S_DATASIZE_20BIT LL_I2S_DATASIZE_24BIT LL_I2S_DATASIZE_32BIT |
| uint32_t rx_threshold | RX FIFO threshold | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> LL_I2S_THRESHOLD_1FIFO LL_I2S_THRESHOLD_2FIFO LL_I2S_THRESHOLD_3FIFO LL_I2S_THRESHOLD_4FIFO LL_I2S_THRESHOLD_5FIFO LL_I2S_THRESHOLD_6FIFO LL_I2S_THRESHOLD_7FIFO LL_I2S_THRESHOLD_8FIFO LL_I2S_THRESHOLD_9FIFO LL_I2S_THRESHOLD_10FIFO LL_I2S_THRESHOLD_11FIFO LL_I2S_THRESHOLD_12FIFO LL_I2S_THRESHOLD_13FIFO LL_I2S_THRESHOLD_14FIFO LL_I2S_THRESHOLD_15FIFO LL_I2S_THRESHOLD_16FIFO |
| uint32_t tx_threshold | TX FIFO threshold | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> LL_I2S_THRESHOLD_1FIFO LL_I2S_THRESHOLD_2FIFO |

| Data Field | Field Description | Value |
|-----------------------|-------------------|---|
| | | <ul style="list-style-type: none"> • LL_I2S_THRESHOLD_3FIFO • LL_I2S_THRESHOLD_4FIFO • LL_I2S_THRESHOLD_5FIFO • LL_I2S_THRESHOLD_6FIFO • LL_I2S_THRESHOLD_7FIFO • LL_I2S_THRESHOLD_8FIFO • LL_I2S_THRESHOLD_9FIFO • LL_I2S_THRESHOLD_10FIFO • LL_I2S_THRESHOLD_11FIFO • LL_I2S_THRESHOLD_12FIFO • LL_I2S_THRESHOLD_13FIFO • LL_I2S_THRESHOLD_14FIFO • LL_I2S_THRESHOLD_15FIFO • LL_I2S_THRESHOLD_16FIFO |
| uint32_t clock_source | Clock source | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • LL_I2S_CLOCK_SRC_96M • LL_I2S_CLOCK_SRC_32M |
| uint32_t audio_freq | Audio frequency | <p>audio_freq = fsck/(2 x wss), in which fsck means the serial clock frequency of I2S and can reach up to 3,027 kHz. WSS can be 16 bits, 24 bits, or 32 bits, depending on the bit width. When the bit width is configured to 16 bits, WSS is 16 bits, and audio_freq can be configured up to 96 kHz.</p> |

3.15.2 I2S Driver APIs

The I2S driver APIs are listed in the table below:

Table 3-78 I2S driver APIs

| API Type | API Name | Description |
|-------------------------------------|----------------------|---|
| Initialization/ Deinitialization | ll_i2s_init() | Initialize I2S. |
| | ll_i2a_deinit() | Deinitialize I2S to default. |
| | ll_i2s_struct_init() | Initialize the structure i2s_init to default. |

The sections below elaborate on these APIs.

3.15.2.1 ll_i2s_init

Table 3-79 ll_i2s_init API

| | |
|-----------------------------|---|
| Function Prototype | error_status_t ll_i2s_init(i2s_regs_t *I2Sx, ll_i2s_init_t *p_i2s_init) |
| Function Description | Initialize I2S peripheral according to specified parameters in ll_i2s_init_t . |
| Parameter | I2Sx: I2S instance p_i2s_init: pointer to the variables of ll_i2s_init_t . The variable contains the configuration information on a specified I2S instance. |
| Return Value | error_status_t shows the enumeration type, which can be: <ul style="list-style-type: none"> • SUCCESS: Initialization of I2S succeeds. • ERROR: Initialization fails. |
| Remarks | |

3.15.2.2 ll_i2s_deinit

Table 3-80 ll_i2s_deinit API

| | |
|-----------------------------|---|
| Function Prototype | error_status_t ll_i2s_deinit(i2s_regs_t *I2Sx) |
| Function Description | Deinitialize I2S to default. |
| Parameter | I2Sx: I2S instance |
| Return Value | error_status_t shows the enumeration type, which can be: <ul style="list-style-type: none"> • SUCCESS: Deinitialization of I2S succeeds. • ERROR: Deinitialization fails. |
| Remarks | |

3.15.2.3 ll_i2s_struct_init

Table 3-81 ll_i2s_struct_init API

| | |
|-----------------------------|---|
| Function Prototype | void ll_i2s_struct_init(ll_i2s_init_t *p_i2s_init) |
| Function Description | Initialize variables of ll_i2s_init_t to default. |
| Parameter | p_i2s_init: pointer to structure variables to be reset |
| Return Value | None |
| Remarks | |

3.16 LL RNG Generic Driver

3.16.1 RNG Driver Structure

3.16.1.1 ll_rng_init_t

The initialization structure `ll_rng_init_t` of the LL RNG driver is defined below:

Table 3-82 `ll_rng_init_t` structure

| Data Field | Field Description | Value |
|---------------------------------|--|--|
| <code>uint32_t seed</code> | Defining linear-feedback shift register (LFSR) seeds | This parameter can be one of the following values: <ul style="list-style-type: none"> • <code>LL_RNG_SEED_FRO_S0</code> (LFSR seed defined by switching oscillator s0) • <code>LL_RNG_SEED_USER</code> (LFSR seed defined by users) |
| <code>uint32_t lfsr_mode</code> | LFSR configuration mode | This parameter can be one of the following values: <ul style="list-style-type: none"> • <code>LL_RNG_LFSR_MODE_59BIT</code> (59-bit LFSR) • <code>LL_RNG_LFSR_MODE_128BIT</code> (128-bit LFSR) |
| <code>uint32_t out_mode</code> | Random number output mode | This parameter can be one of the following values: <ul style="list-style-type: none"> • <code>LL_RNG_OUTPUT_FRO_S0</code> (numbers directly from RNG) • <code>LL_RNG_OUTPUT_CYCLIC_PARITY</code> (cyclic sampling from LFSR and RNG, and odd-even parity generation) • <code>LL_RNG_OUTPUT_CYCLIC</code> (cyclic sampling from LFSR and RNG) • <code>LL_RNG_OUTPUT_LFSR_RNG</code> ($LFSR \oplus RNG$) • <code>LL_RNG_OUTPUT_LFSR</code> (LFSR direct output) <p>Note: When <code>seed_mode</code> is set as <code>LL_RNG_SEED_USER</code>, <code>out_mode</code> cannot be set as <code>LL_RNG_OUTPUT_FRO_S0</code>.</p> |
| <code>uint32_t post_mode</code> | Post-processing mode | This parameter can be one of the following values: <ul style="list-style-type: none"> • <code>LL_RNG_POST_PRO_NOT</code> (no processing) • <code>LL_RNG_POST_PRO_SKIPPING</code> (skipping) • <code>LL_RNG_OUTPUT_CYCLIC</code> (bit counting) • <code>LL_RNG_OUTPUT_LFSR_RNG</code> (Von Neumann architecture) |
| <code>uint32_t interrupt</code> | Interrupt | This parameter can be one of the following values: <ul style="list-style-type: none"> • <code>LL_RNG_IT_DISABLE</code> • <code>LL_RNG_IT_ENABLE</code> |

3.16.2 RNG Driver APIs

The RNG driver APIs are listed in the table below:

Table 3-83 RNG driver APIs

| API Type | API Name | Description |
|---------------------------------|------------------------------|------------------------------|
| Initialization/Deinitialization | <code>ll_rng_init()</code> | Initialize RNG |
| | <code>ll_rng_deinit()</code> | Deinitialize RNG to default. |

| API Type | API Name | Description |
|----------|----------------------|---|
| | ll_rng_struct_init() | Initialize the structure rng_init to default. |

The sections below elaborate on these APIs.

3.16.2.1 ll_rng_init

Table 3-84 ll_rng_init API

| | |
|-----------------------------|---|
| Function Prototype | error_status_t ll_rng_init(rng_regs_t *RNGx, ll_rng_init_t *p_rng_init) |
| Function Description | Initialize RNG peripheral according to specified parameters in ll_rng_init_t . |
| Parameter | RNGx: RNG peripheral instance p_rng_init: pointer to the variables of ll_rng_init_t . The variable contains the configuration information on a specified RNG instance. |
| Return Value | error_status_t shows the enumeration type, which can be: <ul style="list-style-type: none"> • SUCCESS: Initialization of RNG succeeds. • ERROR: Initialization fails. |
| Remarks | |

3.16.2.2 ll_rng_deinit

Table 3-85 ll_rng_deinit API

| | |
|-----------------------------|---|
| Function Prototype | error_status_t ll_rng_deinit(rng_regs_t *RNGx) |
| Function Description | Deinitialize the RNG to default reset values. |
| Parameter | RNGx: RNG peripheral instance |
| Return Value | error_status_t shows the enumeration type, which can be: <ul style="list-style-type: none"> • SUCCESS: Deinitialization of RNG succeeds. • ERROR: Deinitialization fails. |
| Remarks | |

3.16.2.3 ll_rng_struct_init

Table 3-86 ll_rng_struct_init API

| | |
|-----------------------------|---|
| Function Prototype | void ll_rng_struct_init(ll_rng_init_t *p_rng_init); |
| Function Description | Initialize variables of ll_rng_init_t to default. |
| Parameter | p_rng_init: pointer to structure variables to be reset |
| Return Value | None |
| Remarks | |

3.17 LL COMP Generic Driver

3.17.1 COMP Driver Structures

3.17.1.1 ll_comp_init_t

The initialization structure ll_comp_init_t of the COMP driver is defined below:

Table 3-87 ll_comp_init_t structure

| Data Field | Field Description | Value |
|-----------------------|-------------------|--|
| uint32_t input_source | Input source | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • COMP_INPUT_SRC_IO0 (MSIO0) • COMP_INPUT_SRC_IO1 (MSIO1) • COMP_INPUT_SRC_IO2 (MSIO2) • COMP_INPUT_SRC_IO3 (MSIO3) • COMP_INPUT_SRC_IO4 (MSIO4) |
| uint32_t ref_source | Reference source | <p>This parameter can be one of the following values:</p> <ul style="list-style-type: none"> • COMP_REF_SRC_IO0 (MSIO0) • COMP_REF_SRC_IO1 (MSIO1) • COMP_REF_SRC_IO2 (MSIO2) • COMP_REF_SRC_IO3 (MSIO3) • COMP_REF_SRC_IO4 (MSIO4) • COMP_REF_SRC_VBAT • COMP_REF_SRC_VREF |
| uint32_t ref_value | Reference value | <ul style="list-style-type: none"> • If ref_source = COMP_REF_SRC_VBAT, the range of ref_value is 0 to 7. • If ref_source = COMP_REF_SRC_VREF, the range of ref_value is 0 to 63. • If ref_source = COMP_REF_SRC_IOX (X ranges from 0 to 4), the reference value depends on input, and ref_value is invalid. |

3.17.2 COMP Driver APIs

The COMP driver APIs are listed in the table below:

Table 3-88 RNG driver APIs

| API Type | API Name | Description |
|---------------------------------|------------------|-------------------------------|
| Initialization/Deinitialization | ll_comp_init() | Initialize COMP. |
| | ll_comp_deinit() | Deinitialize COMP to default. |

| API Type | API Name | Description |
|----------|-----------------------|---|
| | ll_comp_struct_init() | Initialize the structure ll_comp_init_t to default. |

The sections below elaborate on these APIs.

3.17.2.1 ll_comp_init

Table 3-89 ll_comp_init API

| | |
|-----------------------------|--|
| Function Prototype | error_status_t ll_comp_init(ll_comp_init_t *p_comp_init) |
| Function Description | Initialize COMP peripheral according to specified parameters in ll_comp_init_t. |
| Parameter | p_comp_init: pointer to the variables of ll_comp_init_t. The variable contains the configuration information on a specified COMP. |
| Return Value | error_status_t shows the enumeration type, which can be: <ul style="list-style-type: none"> • SUCCESS: Initialization of COMP succeeds. • ERROR: Initialization fails. |
| Remarks | |

3.17.2.2 ll_comp_deinit

Table 3-90 ll_comp_deinit API

| | |
|-----------------------------|--|
| Function Prototype | error_status_t ll_comp_deinit(void) |
| Function Description | Deinitialize the COMP to default reset values. |
| Parameter | None |
| Return Value | error_status_t shows the enumeration type, which can be: <ul style="list-style-type: none"> • SUCCESS: Deinitialization of COMP succeeds. • ERROR: Deinitialization fails. |
| Remarks | |

3.17.2.3 ll_comp_struct_init

Table 3-91 ll_comp_struct_init API

| | |
|-----------------------------|---|
| Function Prototype | void ll_comp_struct_init(ll_comp_init_t *p_comp_init) |
| Function Description | Initialize variables of ll_comp_init_t to default. |
| Parameter | p_comp_init: pointer to structure variables to be reset |
| Return Value | None |
| Remarks | |

4 Glossary and Abbreviations

Table 4-1 Glossary and abbreviations

| Abbreviation | Description |
|--------------|---|
| ADC | Analog-to-digital Converter |
| AES | Advanced Encryption Standard |
| AON GPIO | Always-on GPIO |
| AON WDT | Always-on WDT |
| API | Application Programming Interface |
| Bluetooth LE | Bluetooth Low Energy |
| DMA | Direct Memory Access |
| GPIO | General Purpose I/O |
| HAL | Hardware Abstraction Layer |
| HMAC | Hash Message Authentication Code |
| I2C | Inter-integrated Circuit |
| I2S | Inter-IC Sound |
| LL | Low Layer |
| MSIO | Mixed Signal I/O |
| MSP | MCU Specific Package |
| NVIC | Nested Vectored Interrupt Controller |
| PKC | Public Key Cipher |
| PWM | Pulse Width Modulation |
| PWR | Power Controller |
| RNG | Random Number Generator |
| SPI | Serial Peripheral Interface |
| SysTick | System Tick Timer |
| TIM | Timer |
| UART | Universal Asynchronous Receiver/Transmitter |
| WDT | Watchdog Timer |