



32-Bit Arm® Cortex®-M33/M0+
Dual Core Bluetooth® Low Energy Microcontroller

HT32F67595

User Manual

Revision: V1.00 Date: May 12, 2025

[**www.holtek.com**](http://www.holtek.com)

Table of Contents

1 Power Management Control Unit (PMU)	8
Introduction	8
Features	9
Functional Description	9
Power Management	9
Low Power Management	9
PMU APIs	12
PMU Interrupt APIs	12
PMU Power APIs	15
PMU Clock APIs	21
PMU Example	28
PMU Example Code	28
2 General Purpose I/O (GPIO)	29
Introduction	29
Features	29
Functional Description	29
Input Mode	30
Output Mode	30
Drive Capability	30
Function Reuse	31
Interrupt Function	31
Wakeup Function	31
GPIO APIs	31
GPIO ROM APIs	31
SW Port API Function	39
IR API Function	42
DCXO Pins API Function	44
GPIO Examples	45
Example Code for Input	45
Example Code for Output	45
Example Code for GPIO Interrupt	45
Example Code for GPIO Wakeup	46
3 General-Purpose Timer (GPTM)	47
Introduction	47
Features	47
Functional Description	47
Counter Mode (CNT Mode)	47
Capture Mode	49
PWM Mode	51

GPTM APIs	51
Timer Function APIs.....	51
PWM Function APIs.....	63
Timer Capture Function	67
GPTM Examples.....	72
Counter Mode Example Code	72
PWM Mode Example Code	73
4 System Tick Timer (STIM)	74
Introduction	74
Features.....	74
Functional Description	74
Clock Accuracy and Prescale	75
Counter, Tick, Overflow and Compare.....	75
STIM APIs	76
STIM Interrupt APIs	76
STIM Wakeup & Status APIs	78
STIM Counter & Compare	80
STIM Example Code.....	82
STIM Wakeup & Interrupt Example Code.....	82
STIM Overflow Interrupt Example Code	83
5 Real Time Clock (RTC)	84
Introduction	84
Features.....	84
Functional Description	84
RTC Power Structure.....	85
Clock Accuracy and Prescale	86
Counter, Tick, Overflow and Compare.....	86
RTC APIs	87
RCT Configuration APIs.....	87
RTC Interrupt APIs.....	88
RTC Status APIs.....	91
RTC Counter & Compare	92
RTC Example Code	95
RTC Start Work Example Code.....	95
RTC Interrupt Example Code	95
Wakeup Example Code	96
6 Universal Asynchronous Receiver Transmitter (UART).....	97
Introduction	97
Features.....	97
Functional Description	97

UART APIs	97
UART Configure APIs	97
UART Interrupt APIs	101
UART FIFO APIs	103
UART RX/TX APIs	105
UART Example Code	106
Example of Application Using UART1	106
7 IR Module	109
Introduction	109
Features	109
Functional Description	109
IR Encode	109
IR Decode	110
IR Module APIs	111
IR Encode HW APIs	111
IR Encode HAL APIs	114
IR Decode HW APIs	116
IR Decode HAL APIs	118
IR Module Examples	120
IR NEC Encode Example	120
IR Decode Example	122
8 2nd-Boot	123
Memory Map	123
The 2nd-Boot Code	124
OTA Flow	124
Initialize Cache Read	126
Jump to SDK	126
9 IPC	127
Introduction	127
IPC API Introduction	127
IPC APIs Based on Message Queue Implementation	127
IPC APIs Based on Array Implementation	128
IPC Example Code	129
IPC Initialization (MP is the same as CP)	129
IPC Message Receiving Callback Function	129
IPC Message Sending Function	129
10 Bluetooth Low Energy GAP	131
Introduction	131
Bluetooth Low Energy GAP API Interfaces	131
General Interfaces	131

Advertising Interfaces	134
Scanning interfaces	137
Initiating State Interfaces	139
Connection State Interfaces	141
Pairing and Encrypted Connection Related Interfaces	146
Privacy Function Related Interfaces	152
Stack Messages	155
Examples	162
General Interfaces	162
Advertising	162
Scanning	164
Peripheral	166
Central	171
11 Bluetooth Low Energy GATT	173
Introduction	173
Bluetooth Low Energy GATT Server API Interfaces	173
API Interfaces	173
Characteristic Property Description	179
Transaction Mode Description in Write Characteristic Callback Function	180
Example Code: Transparent Profile	180
Bluetooth Low Energy GATT Client API Interfaces	191
API Interfaces	191
Callback Function	201
Structure Definition	202
Error Code	203
Example Code	204
Appendix	214
Common UUID References	214

List of Tables

Table 1. LUT Table	11
Table 2. Look-up Table Event ID	11
Table 3. GPTM Mode Selection	48
Table 4. Time Interval.....	75
Table 5. Memory Map	123

List of Figures

Figure 1. PMU Block Diagram	8
Figure 2. GPIO Basic Structure Diagram.....	29
Figure 3. GPTM Counter Register	48
Figure 4. GPTM Prescaler	48
Figure 5. Timing of GPTM Work Mode	49
Figure 6. STIM Reference Design Schematic	74
Figure 7. Counter Trigger Overflow	75
Figure 8. RTC Reference Design Schematic.....	85
Figure 9. RTC Power Reference Design Schematic.....	85
Figure 10. Counter Work Process.....	86
Figure 11. Counter Trigger Overflow.....	86
Figure 12. IR Encode.....	109
Figure 13. IR Decode.....	110
Figure 14. Lead Code	121
Figure 15. Bit Description	121
Figure 16. Repetition Code.....	121
Figure 17. Start-up Flow	123
Figure 18. The 2nd-Boot Code Flow.....	124
Figure 19. Flash Memory Map.....	124
Figure 20. OTA Process in 2nd-Boot	125
Figure 21. Initialize Cache	126

1 Power Management Control Unit (PMU)

Introduction

The HT32F67595 device has a completely integrated Power Management Unit (PMU). This includes a Single Inductance Single Output (SISO) DC-DC converter with one output, and a number of LDOs for the different power rails of the system.

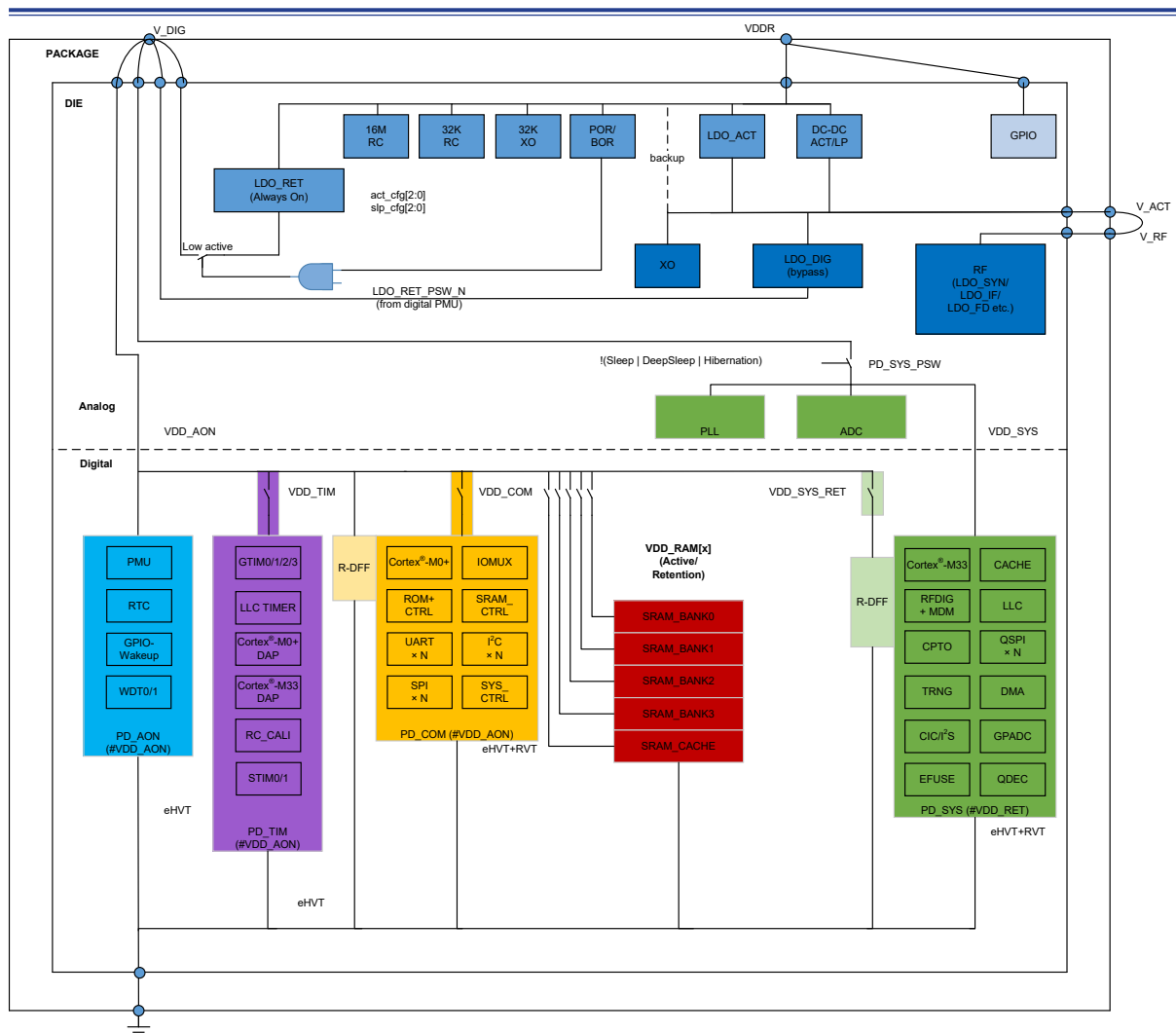


Figure 1. PMU Block Diagram

Features

- Support a number of LDOs:
 - LDO_ACT with 0.95 V ~ 1.30 V configurable
 - LDO_DIG with 0.9 V ~ 1.25 V configurable
 - LDO_RET with 0.75 V ~ 1.1 V configurable
- Support a SISO DC-DC:
 - DCDC_ACT with 0.95 V ~ 1.30 V configurable
 - DCDC_RET with 0.65 V ~ 1.05 V configurable
 - DCDC_RET clock 1 / 2 / 4 ... / 128 division configurable
- Multiple clocks
 - System clock: RC_HCLK (RC16M/RC48M), DCXO16M, PLL64M configurable
 - Low power clock: RC32K, DCXO32K configurable
- Four system working modes:
 - Active Mode: The system works in normal mode. Different power supplies and system clocks (RC_HCLK/DCXO_HCLK/PLL) can be configured
 - Sleep Mode: The system clock is powered off, the SRAM is keeping retention
 - Deep-Sleep Mode: The SRAM is powered off, the low power clock (32K) is keeping retention
 - Hibernation Mode: The low power clock (32K) is powered off
- Support each SRAM block retention configurable when the system get into sleep
 - Support 16 Look-up Table (LUT) for wakeups
 - Support power monitoring unit, POR and BOR

Functional Description

Power Management

The device supports two power modes: LDO and DC-DC. The system works in the LDO mode by default. The power mode switching takes effect only after the chip is woken up. In the DC-DC mode, an external inductor is required, but the system power consumption is lower than that in the LDO mode.

Low Power Management

The working conditions of each module in four working modes are as follows:

	Active	Sleep	Deep-Sleep	Hibernation
CPU	Active	Off	Off	Off
FLASH	On	Available	Off	Off
SRAM	On	On	Off	Off
RADIO	On	Off	Off	Off
SRAM Retention	Full	Partial	No	No
16M RC / DCXO	On	Off	Off	Off

	Active	Sleep	Deep-Sleep	Hibernation
32K RC / DCXO	On	On	On	Off
Peripheral	Available	Available	Off	Off
Wake up on RTC	Available	Available	Available	OFF
Wake up on GPIO	Available	Available	Available	Available

In the Active mode, the application Arm® Cortex®-M33 or Arm® Cortex®-M0+ CPU is actively executing code. The Active mode provides normal operation of the processor and all of the peripherals that are currently enabled. The system clock can be any available clock source.

In the Sleep mode, all active peripheral can be clocked. But the application CPU core and memory are not clocked and no code is executed. Any interrupt event will bring the processor back into active mode.

In the Deep-Sleep mode, only the always-on domain is active. An external wakeup event RTC event is required to bring the device back to active mode.

In the Hibernation mode, the device is turned off entirely, including the always-on domain. The I/Os are latched with the value they had before entering hibernation mode. A change of state on any I/O pin defined as a wakeup from hibernation pin wakes up the device and functions as a reset trigger.

Get into Low Power Mode

Since the device is a dual-core MCU, the Cortex®-M33 and Cortex®-M0+ must get into sleep before the chip starts to get into sleep process. The Cortex®-M0+ is configured in the low power mode by default. Users mainly configure low power in Cortex®-M33.

In main.c, call the `lpwr_ctrl_init` function for low power initialization, as follows:

```
lpwr_ctrl_init(enMode, lpwr_before_sleep, lpwr_after_wakeup);
```

`enMode` refers to `EN_LPWR_MODE_SEL_T`, as follows:

```
typedef enum
{
    LPWR_MODE_ACTIVE = 0x00,
    LPWR_MODE_IDLE = 0x01,
    LPWR_MODE_SLEEP = 0x02, /* SRAM keep retention, system clock will power down. */
    LPWR_MODE_DEEPSLEEP = 0x03, /* SRAM will power down, LCLK clock will keep
                                working(shutdown with 32K). */
    LPWR_MODE_HIBERNATION = 0x04, /* SRAM will power down, LCLK clock will turn
                                off(shutdown without 32K). */
} EN_LPWR_MODE_SEL_T;
```

Note: 1. If `enMode` is set to `LPWR_MODE_ACTIVE`, the CPU will keep active, the system clock will not be turned off.

2. If `enMode` is set to `LPWR_MODE_IDLE`, the CPU will keep idle mode, the system clock will not be turned off.

- `lpwr_before_sleep` is a callback function to check if the system can get into sleep. Users can add application code here to determine the system can get into sleep or not. The system will not get into sleep when return 0, otherwise it will get into sleep mode.
- `lpwr_after_wakeup` is a callback function which is the first to be executed after the system is woken up from the sleep mode. Users can add application code here to reinitialize peripherals.
- When `lpwr_ctrl_goto_sleep()` was executed, the system will get into the low power mode of `enMode` defined.

System Wakeup

The device PMU module has a 13-bit word Look-up Table (LUT). It defines maximum 16 events, which will wake up the system from the low power mode (Sleep / Deep-Sleep or Hibernation Mode) when happened.

The Look-up Table (LUT) is initialized at cold boot by the Cortex®-M33. It instructs the PDC which digital power domains to activate based on the triggering source.

Table 1. LUT Table

	Bit	Function			
Triggers	0	If 0x00 then	If 0x01 then	If 0x02 then	If 0x03 then
	1	P0 GPIO	P1 GPIO	P2 GPIO	Peripheral
	2	Pin_ID	Pin_ID	Pin_ID	Periph_ID0
	3	Pin_ID	Pin_ID	Pin_ID	Periph_ID1
	4	Pin_ID	Pin_ID	Pin_ID	Periph_ID2
	5	Pin_ID	Pin_ID	Pin_ID	Periph_ID3
	6	Pin_ID	Pin_ID	Pin_ID	Periph_ID4
What to Do	7	Enable PD_SYS			
	8	Enable DCXO16M			
	9	Enable Send Interrupt to Cortex®-M0+			
	10	Enable Send Interrupt to Cortex®-M33			

The depth of the LUT is 16 places, so the system supports up to 16 different wakeup configurations, but could be changed dynamically by application software.

Four different trigger types are evaluated when a trigger comes according to the value on bit 0 and bit 1 of each LUT entry:

- If Bits [1:0] = 00 then it is a GPIO toggle from P0. Bits [6:2] contain the pin number that is triggered;
- If Bits [1:0] = 01 then it is a GPIO toggle from P1. Bits [6:2] contain the pin number that is triggered;
- If Bits [1:0] = 10 then it is a GPIO toggle from P2. Bits [6:2] contain the pin number that is triggered;
- If Bits [1:0] = 11 then it is a trigger from some peripheral. Bits [6:2] define the peripheral according to follows:

Table 2. Look-up Table Event ID

ID (Decimal)	Peripheral
0	LUT_TRIG_ID_RTC_CH0
1	LUT_TRIG_ID_RTC_CH1
2	LUT_TRIG_ID_RTC_CH2
3	LUT_TRIG_ID_RTC_CH3
4	LUT_TRIG_ID_LLC
5	LUT_TRIG_ID_GTIM0
6	LUT_TRIG_ID_GTIM1
7	LUT_TRIG_ID_GTIM2
8	LUT_TRIG_ID_GTIM3
9	LUT_TRIG_ID_GPIO
10	LUT_TRIG_ID_CP_SWD

ID (Decimal)	Peripheral
11	LUT_TRIG_ID_MP_SWD
12	Reserved
13	LUT_TRIG_ID_STIM0_CH0
14	LUT_TRIG_ID_STIM0_CH1
15	LUT_TRIG_ID_STIM0_CH2
16	LUT_TRIG_ID_STIM0_CH3
17	LUT_TRIG_ID_STIM1_CH0
18	LUT_TRIG_ID_STIM1_CH1
19	LUT_TRIG_ID_STIM1_CH2
20	LUT_TRIG_ID_STIM1_CH3
21	LUT_TRIG_ID_WDT1

Note: The ID 9 indicates that all GPIOs wake up with debounce, only in Sleep and Deep-Sleep can set.

Bits [1:0] = 00/01/10 indicates that all GPIOs wake up without debounce.

Bits [10:7] explain what needs to be done upon a trigger from the triggering sources as explained so far. More specifically, a triggering source might request to:

- Enable DCXO16M. This bit is set if the clock precision is required by the application (for example, Bluetooth® Low Energy operation, high-performance mode which needs the PLL64).
- Enable Send Interrupt to Cortex®-M0+ / Cortex®-M33. The Peripheral PMU interrupt will be sent to Cortex®-M0+ or Cortex®-M33.

PMU APIs

PMU Interrupt APIs

rom_hw_pmu_get_interrupt_flag

Function Prototype

```
EN_ERR_STA_T rom_hw_pmu_get_interrupt_flag (uint32_t* pu32IntMsk);
```

Description

Get the PMU interrupt flag (status) by reading the PMU_LUT_INT_FLAG register.

Parameter

Parameter	Description
pu32IntMsk	Indicate which interrupt flag will be read.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_pmu_clear_interrupt_flag

Function Prototype

EN_ERR_STA_T rom_hw_pmu_clear_interrupt_flag (uint32_t u32IntMsk);

Description

Clear the PMU interrupt flag (status) by writing the PMU_LUT_INT_CLR register.

Parameter

Parameter	Description
u32IntMsk	Indicate which flag will be cleared.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_pmu_set_wakeup_source

Function Prototype

EN_ERR_STA_T rom_hw_pmu_set_wakeup_source (uint8_t u8LutIdx, PUM_LUT_TRIG_SEL_T enTrigSel, EN_LUT_TRIG_ID_T enLutTrigId, EN_LUT_ACT_T enLutAction);

Description

Select the PMU wakeup source.

Parameter

Parameter	Description
u8LutIdx	LUT index, need < 16.
enTrigSel	LUT trigger selection, @ ref PUM_LUT_TRIG_SEL_T.
enLutTrigId	LUT trigger ID selection, @ ref EN_LUT_TRIG_ID_T.
enLutAction	LUT action selection, @ ref EN_LUT_ACT_T.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_pmu_set_gpio_wakeup_source

Function Prototype

EN_ERR_STA_T rom_hw_pmu_set_gpio_wakeup_source (uint8_t u8LutIdx, PUM_LUT_TRIG_SEL_T enTrigSel, uint32_t u32Pin, EN_LUT_ACT_T enLutAction)

Description

Set GPIOs wake up system from low power mode without debounce.

Parameter

Parameter	Description
u8LutIdx	LUT index, need < 16.
enTrigSel	LUT trigger configuration, must select PMU_LUT_TRIG_GPIOA, PMU_LUT_TRIG_GPIOB, PMU_LUT_TRIG_GPIOC.

Parameter	Description
u32Pin	Which pin needs to be configured, only one pin can be configured at the same time.
u8LutAction	LUT action configuration, @ ref EN_LUT_ACT_T.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_pmu_set_sram_block_ret

Function Prototype

EN_ERR_STA_T rom_hw_pmu_set_sram_block_ret (uint32_t u32SramBlock);

Description

Configure the SRAM memory keeping retention when the system gets into the low power mode.

Parameter

Parameter	Description
u32SramBlock	Blocks of RAM which will keep retention. Each bit corresponding a SRAM block.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_pmu_clr_sram_block_ret

Function Prototype

EN_ERR_STA_T rom_hw_pmu_clr_sram_block_ret (uint32_t u32SramBlock);

Description

Clear the indicated SRAM block retention function when the system gets into the Sleep mode.

Parameter

Parameter	Description
u32SramBlock	Blocks of ram which will shut down. Each bit corresponding a SRAM block.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

PMU Power APIs

rom_hw_pmu_enable_ldo_act_output

Function Prototype

EN_ERR_STA_T rom_hw_pmu_enable_ldo_act_output (void);

Description

Enable the LDO_ATC output.

The LDO_ACT output supplies the CPU power when the system works in the Active mode.

Parameter

None.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_pmu_disable_ldo_act_output

Function Prototype

EN_ERR_STA_T rom_hw_pmu_disable_ldo_act_output (void);

Description

Disable the LDO_ATC output.

See rom_hw_pmu_enable_ldo_act_output.

Parameter

None.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_pmu_set_ldo_act_voltage

Function Prototype

EN_ERR_STA_T rom_hw_pmu_set_ldo_act_voltage (EN_LDO_ACT_VOLT_T enVolt);

Description

Configure the LDO_ACT output voltage, it is connected to VDD_RF to supply the RF module when the system works in the Active mode.

Parameter

Parameter	Description
enVolt	Refer to EN_LDO_ACT_VOLT_T. The range is 950 mV ~ 1300 mV by 50 mV step. Note: The LDO_ACT output must be at least 50 mV higher than the voltage of LDO_DIG output.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_pmu_enable_ldo_dig_output

Function Prototype

EN_ERR_STA_T rom_hw_pmu_enable_ldo_dig_output (void);

Description

Enable the LDO_DIG output.

The LDO_DIG output supplies the digital module when the system works in the Active mode.

Parameter

None.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_pmu_disable_ldo_dig_output

Function Prototype

EN_ERR_STA_T rom_hw_pmu_disable_ldo_dig_output (void);

Description

Disable the LDO_DIG output.

See rom_hw_pmu_enable_ldo_dig_output.

Parameter

None.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_pmu_set_ldo_dig_voltage

Function Prototype

EN_ERR_STA_T rom_hw_pmu_set_ldo_dig_voltage (EN_LDO_DIG_VOLT_T enVolt);

Description

Configure the LDO_DIG output voltage.

See rom_hw_pmu_enable_ldo_dig_output.

Parameter

Parameter	Description
enVolt	Voltage of LDO_DIG definition. The range is 900 mV ~ 1100 mV by 50 mV step. Note: 1. The LDO_DIG output voltage could set to 900 mV when the SYS_CLK works in 16 MHz or less. 2. The LDO_DIG output voltage must higher than 1100 mV when the SYS_CLK works in 128 MHz.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_pmu_get_ldo_dig_voltage

Function Prototype

EN_ERR_STA_T rom_hw_pmu_get_ldo_dig_voltage (uint8_t* pu8Volt);

Description

Get the current LDO_DIG output voltage configuration.

Parameter

Parameter	Description
enVolt	Get the LDO Output Voltage, @ ref EN_LDO_DIG_VOLT_T.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_pmu_set_ldo_dig_and_ret_act_voltage

Function Prototype

EN_ERR_STA_T rom_hw_pmu_set_ldo_dig_and_ret_act_voltage (EN_LDO_DIG_VOLT_T enVolt);

Description

Configure the LDO_DIG, LDO_RET and LDC_ACT output voltage at the same time.

Parameter

Parameter	Description
enVolt	The LDO_DIG, LDO_RET and LDO_ACT output voltage, the range is 900 mV ~ 1100 mV by 50 mV Step. See LDO_DIG, LDO_RET and LDO_ACT for details.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_pmu_set_ldo_ret_act_voltage

Function Prototype

```
EN_ERR_STA_T rom_hw_pmu_set_ldo_ret_act_voltage (EN_LDO_RET_VOLT_T enVolt);
```

Description

Configure the LDO_RET output voltage when the system works in the Active mode.

Parameter

Parameter	Description
enVolt	The LDO_RET output voltage when the system works in the Active mode. The range is 750 mV ~ 1100 mV by 50 mV step. Note: Please keep the LDO_RET output the same voltage with LDO_DIG when the system works in the Active mode.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_pmu_set_ldo_ret_sleep_voltage

Function Prototype

```
EN_ERR_STA_T rom_hw_pmu_set_ldo_ret_sleep_voltage (EN_LDO_RET_VOLT_T enVolt);
```

Description

Configure the LDO_RET output voltage when the system works in the SLEEP mode.

Parameter

Parameter	Description
enSleepVolt	Configure the LDO_RET output voltage when the system works in the SLEEP mode. The range is 750 mV to 1100 mV by 50 mV step.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_pmu_dcdc_init

Function Prototype

```
EN_ERR_STA_T rom_hw_pmu_dcdc_init (void);
```

Description

Initialize the DC-DC module.

Parameter

None.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_pmu_enable_dcdc_act_output

Function Prototype

EN_ERR_STA_T rom_hw_pmu_enable_dcdc_act_output (void);

Description

Enable the DC-DC output when the system works in the Active mode.

Parameter

None.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_pmu_disable_dcdc_act_output

Function Prototype

EN_ERR_STA_T rom_hw_pmu_disable_dcdc_act_output (void);

Description

Enable the indicated DMA controller.

Parameter

None.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_pmu_set_dcdc_act_voltage

Function Prototype

EN_ERR_STA_T rom_hw_pmu_set_dcdc_act_voltage (EN_DCDC_ACT_VOLT_T enVolt);

Description

Configure the DC-DC module output voltage when the system works in the Active mode.

The DC-DC module supplies the power of the RF module and digital module in the DC-DC mode.

Parameter

Parameter	Description
enVolt	The DC-DC output voltage. Its range is 950 mV ~ 1300 mV by 50 mV step. Note: The DC-DC output voltage must be 50 mV higher than LDO_DIG output voltage at least.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_pmu_enable_dcdc_ret_output

Function Prototype

EN_ERR_STA_T rom_hw_pmu_enable_dcdc_ret_output (void);

Description

Enable the DC-DC module output when the system works in the low power mode including Sleep, Deep-Sleep and Hibernation mode.

Parameter

None.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_pmu_disable_dcdc_ret_output

Function Prototype

EN_ERR_STA_T rom_hw_pmu_disable_dcdc_ret_output (void);

Description

Disable the DC-DC module output when the system works in the low power mode including Sleep, Deep-Sleep and Hibernation mode.

Parameter

None.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_pmu_set_dcdc_ret_voltage

Function Prototype

EN_ERR_STA_T rom_hw_pmu_set_dcdc_ret_voltage (EN_DCDC_LPWR_VOLT_T enLpwrVolt);

Description

Configure the DC-DC output voltage when the system works in the low power mode including Sleep, Deep-Sleep and Hibernation mode.

Parameter

Parameter	Description
enLpwrVolt	The DC-DC module output voltage when the system works in the low power mode. Its range is 650 mV to 1050 mV by 50 mV step.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

PMU Clock APIs

rom_hw_pmu_set_dcdc_ret_clk_divisor

Function Prototype

```
EN_ERR_STA_T rom_hw_pmu_set_dcdc_ret_clk_divisor (EN_DCDC_LPWR_CLK_T enDiv);
```

Description

Configure the DC-DC module clock division value from the internal RC_CLK. The DC-DC module clock is sourced from the internal RC_CLK when the system works in the low power mode including Sleep, Deep-Sleep and Hibernation mode.

Parameter

Parameter	Description
enDiv	The division value from the internal RC_CLK.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_pmu_set_low_power_mode

Function Prototype

```
EN_ERR_STA_T rom_hw_pmu_set_low_power_mode (EN_PWR_MODE_T enMode);
```

Description

Set the system to get into the indicated low power mode including Sleep, Deep-Sleep and Hibernation mode.

Parameter

Parameter	Description
enMode	Low power working mode. @ ref EN_PWR_MODE_T.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_pmu_rc_lclk_is_enable

Function Prototype

```
bool rom_hw_pmu_rc_lclk_is_enable (void);
```

Description

Check whether the internal RC_CLK is enabled or not.

Parameter

None.

Return

status	FALSE(0) – RC_LCLK is disabled; TURE(1) – RC_LCLK is enabled.
--------	--

rom_hw_pmu_dcxo_lclk_is_power_on

Function Prototype

```
bool rom_hw_pmu_dcxo_lclk_is_power_on (void);
```

Description

Check whether the DCXO Low Clock (DCXO_LCLK) is powered on or off.

Parameter

None.

Return

status	FALSE(0) – DCXO_LCLK is powered on; TURE(1) – DCXO_LCLK is powered off.
--------	--

rom_hw_pmu_dcxo_lclk_is_clk_out

Function Prototype

```
bool rom_hw_pmu_dcxo_lclk_is_clk_out (void);
```

Description

Check whether the DCXO Low Clock (DCXO_LCLK) output is enabled or not.

Parameter

None.

Return

status	FALSE(0) – DCXO_LCLK output is disabled; TRUE(1) – DCXO_LCLK output is enabled.
--------	--

rom_hw_pmu_rc_hclk_is_power_on

Function Prototype

```
bool rom_hw_pmu_rc_hclk_is_power_on (void);
```

Description

Check whether the RC High Clock (RC_HCLK) is powered on or off.

Parameter

None.

Return

status	FALSE(0) – RC_HCLK is powered off; TRUE(1) – RC_HCLK is powered on.
--------	--

rom_hw_pmu_dcxo_hclk_is_clk_out

Function Prototype

```
bool rom_hw_pmu_dcxo_hclk_is_clk_out (void);
```

Description

Check whether the DCXO High Clock (DCXO_HCLK) output is enabled or disabled.

Parameter

None.

Return

status	FALSE(0) – DCXO_HCLK output is disabled; TURE(1) – DCXO_HCLK output is enabled.
--------	--

rom_hw_pmu_dcxo_hclk_is_power_on

Function Prototype

```
bool rom_hw_pmu_dcxo_hclk_is_power_on (void);
```

Description

Check whether the DCXO High Clock (DCXO_HCLK) is powered on or off.

Parameter

None.

Return

status	FALSE(0) – DCXO_HCLK is powered off; TRUE (1) – DCXO_HCLK is powered on.
--------	---

rom_hw_pmu_rc_hclk_is_clk_out

Function Prototype

```
bool rom_hw_pmu_rc_hclk_is_clk_out (void);
```

Description

Check whether the internal high RC Clock (RC_HCLK) output is enabled or disabled.

Parameter

None.

Return

status	FALSE(0) – RC_HCLK output is disabled; TURE(1) – RC_HCLK output is enabled.
--------	--

rom_hw_pmu_pll_clk_is_power_on

Function Prototype

```
bool rom_hw_pmu_pll_clk_is_power_on (void);
```

Description

Check whether the PLL Clock (PLL_CLK) is powered on or off.

Parameter

None.

Return

status	FALSE(0) – PLL_CLK is powered off; TURE(1) – PLL_CLK is powered on.
--------	--

rom_hw_pmu_pll_clk_is_clk_out

Function Prototype

```
bool rom_hw_pmu_pll_clk_is_clk_out(void);
```

Description

Check whether the PLL Clock (PLL_CLK) output is enabled or disabled.

Parameter

None.

Return

status	FALSE(0) – PLL_CLK output is disabled; TRUE (1) – PLL_CLK output is enabled.
--------	---

rom_hw_pmu_turn_clk_power_on

Function Prototype

```
EN_ERR_STA_T rom_hw_pmu_turn_clk_power_on (EN_CLK_POWER_CTRL_T enCLK,  
uint32_t u32TimeUs);
```

Description

Turn on the indicated clock power.

Parameter

Parameter	Description
enCLK	Indicate which clock will be powered on, @ ref EN_CLK_POWER_CTRL_T.
u32TimeUs	The setup time of clock. Unit: μs.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_pmu_turn_clk_power_off

Function Prototype

```
EN_ERR_STA_T rom_hw_pmu_turn_clk_power_off (EN_CLK_POWER_CTRL_T enCLK);
```

Description

Turn off the indicated clock power.

Parameter

Parameter	Description
enCLK	Indicate which clock will be powered off, @ ref EN_CLK_POWER_CTRL_T.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_pmu_enable_clk_output

Function Prototype

EN_ERR_STA_T rom_hw_pmu_enable_clk_output (EN_CLK_OUT_CTRL_T enCLK);

Description

Enable the indicated clock output.

Parameter

Parameter	Description
enCLK	Indicate which clock output will be enabled, @ ref EN_CLK_OUT_CTRL_T.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_pmu_disable_clk_output

Function Prototype

EN_ERR_STA_T rom_hw_pmu_disable_clk_output (EN_CLK_OUT_CTRL_T enCLK);

Description

Disable the indicated clock output.

Parameter

Parameter	Description
enCLK	Indicate which clock output will be disabled, @ ref EN_CLK_OUT_CTRL_T.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_pmu_dcxo_lclk_cfg_is_valid

Function Prototype

bool rom_hw_pmu_dcxo_lclk_cfg_is_valid (stDCXOBuf_t* pstBuf, stDCXOParam_t* pstParam);

Description

Check whether the DCXO Low Clock (DCXO_LCLK) configuration is valid.

Parameter

Parameter	Description
pstBuf	Pointer to a stDCXOBuf_t structure.
pstParam	Pointer to a stDCXOParam_t structure.

Return

status	FALSE(0) – DCXO_LCLK configuration is invalid; TURE(1) – DCXO_LCLK configuration is valid.
--------	---

rom_hw_pmu_set_dcxo_lclk_param

Function Prototype

```
EN_ERR_STA_T rom_hw_pmu_set_dcxo_lclk_param (stDCXOBuf_t* pstBuf, stDCXOParam_t* pstParam);
```

Description

Configure the DCXO Low Clock (DCXO_LCLK, 32 kHz) parameters including inverter, current and load capacitance.

Parameter

Parameter	Description
pstBuf	Pointer to a stDCXOBuf_t structure that contains: u8PosBuf, u8NegBuf: Number of inverters. The range of value is [0:7].
pstParam	Pointer to a stDCXOParam_t structure that contains: u8Ib: DCXO current. The range of value is [0:7]. u8Ngm: The number of inverters in parallel with the active part of the oscillator. The range of value is [0:7]. Suggest cfg to 2 ~ 4. u8Cap: DCXO load capacitance. Unit: 0.1 pF, the range is [0:255] which means 3.0 pF ~ 28.5 pF.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_pmu_set_rc_lclk_tune

Function Prototype

```
EN_ERR_STA_T rom_hw_pmu_set_rc_lclk_tune (uint8_t u8Val);
```

Description

Configure the RC_LCLK calibration value to calibrate the RC_LCLK.

Parameter

Parameter	Description
u8Val	RC_LCLK tune value, the range is [0:255].

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_pmu_set_rc_hclk_tune

Function Prototype

```
EN_ERR_STA_T rom_hw_pmu_set_rc_hclk_tune (uint8_t u8Val);
```

Description

Configure the calibration value to calibrate the RC_HCLK.

Parameter

Parameter	Description
u8Val	RC_HCLK tune value, the range is [0:127].

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_pmu_set_rc_hclk_sync_out

Function Prototype

EN_ERR_STA_T rom_hw_pmu_set_rc_hclk_sync_out (EN_RC_HCLK_SYNC_OUT_MODE_T enMode);

Description

Enable or disable the RC_HCLK clock output while turning on the RC_HCLK.

Parameter

Parameter	Description
enMode	RC_HCLK clock output mode. b0 – Delay 0.5 to 1.5 32K clock cycles to output RC_HCLK after turning on RC_HCLK. b1 – Output RC_HCLK while turning on RC_HCLK immediately.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_pmu_sel_dcxo_hclk_pwr

Function Prototype

EN_ERR_STA_T rom_hw_pmu_sel_dcxo_hclk_pwr (EN_DCXO_HCLK_PWR_T enPwr);

Description

Configure the DCXO High Clock (DCXO_HCLK) power supply to DC-DC or VDDR.

Parameter

Parameter	Description
enPwr	DCXO_HCLK power selection, @ ref EN_DCXO_HCLK_PWR_T.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_pmu_set_dcxo_hclk_stable_time

Function Prototype

EN_ERR_STA_T rom_hw_pmu_set_dcxo_hclk_stable_time (uint8_t u8Time);

Description

Configure the DCXO High Clock (DCXO_HCLK) stable time after wakeup, the unit is 31.25 μ s.

Parameter

Parameter	Description
u8Time	The stable time, the range is [0:255], the unit is 31.25 μ s.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_pmu_set_dcxo_hclk_param

Function Prototype

EN_ERR_STA_T rom_hw_pmu_set_dcxo_hclk_param (stDCXOParam_t* pstParam);

Description

Configure DCXO High Clock (DCXO_HCLK) current and load capacitance.

Parameter

Parameter	Description
pstParam	Pointer to a stDCXOParam_t structure that contains: u8Ib: DCXO current. The range of value is [0:7]. u8Ngm: The number of inverters in parallel with the active part of the oscillator. The range of value is [0:7]. Suggest cfg to 2 ~ 4. u8Cap: DCXO load capacitance. Unit: 0.1 pF, the range is [0:255] which means 3.0 pF ~ 28.5 pF.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

PMU Example

PMU Example Code

```
/**
 * @ brief System power manage.
 * @ param enSel: Select DCDC or LDO, @ ref EN_PMU_POWER_SEL_T.
 */
static void system_power_init(EN_PMU_PWR_SEL_T enSel)
{
    //Set ldo_act voltage.
    rom_hw_pmu_set_ldo_act_voltage(EN_LDO_ACT_1200mV);
    // Init dcdc configuration and set dcdc_act voltage.
    patch_hw_pmu_dcdc_init();
    rom_hw_pmu_set_dcdc_act_voltage(EN_DCDC_ACT_VOLT_1200mV);
    // Set ldo_dig and ldo_ret voltage.
    rom_hw_pmu_set_ldo_dig_and_ret_act_voltage(EN_LDO_DIG_1100mV);
    rom_hw_pmu_set_ldo_ret_sleep_voltage(EN_LDO_RET_1100mV);
    // Power selection. It will be valid after the system
    // gets into sleep, default is ldo mode.
    rom_hal_pmu_sel_power_act_out_mode(enSel);
}
```

2 General Purpose I/O (GPIO)

Introduction

GPIO is short for General Purpose Input and Output port, which can be controlled by software for its input and output.

Features

- Support up to 16 I/O pins
- Output state: floating + pull-up/down
- Input state: Schmidt/CMOS + pull-up/down
- Each I/O supports individual high/low level setting
- Each I/O supports reading the current level status / the output status

Functional Description

Any of the GPIOs in the device supports software configuration for input and output functions, multiplexing for other peripheral interfaces, configuration for external interrupt functions, and can be configured for CPU sleep wakeup.

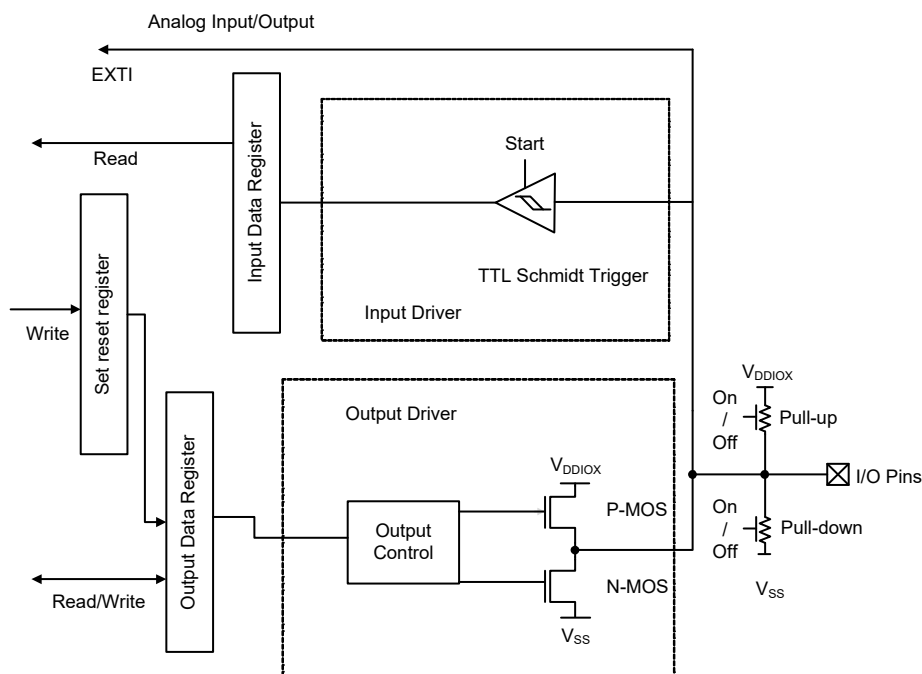


Figure 2. GPIO Basic Structure Diagram

As shown above, the functions of the module are mainly realized through different input and output configurations. Each I/O port of the device can be configured with the corresponding mode of I/O through software, mainly with the following configuration modes:

Input	Input Float	Input Pull-up	Input Pull-down	Analog Input	Schmitt TTL Input	Schmitt CMOS Input
Output	Output Pull-up	Output Pull-down				

Input Mode

Each I/O of the device can be used as an input, and when the I/O port is configured as an input:

- Output buffers are disabled
- Schmitt trigger is turned on
- Pull-up and pull-down resistors are turned on or off depending on the register status
- The input data register samples the data on the I/O pins every 2 APB clock cycles
- I/O status acquisition via input data registers

Output Mode

Each I/O of the device outputs a different level and reads the output status, and when the I/O port is configured as an output:

- Output buffer is turned on
 - When a high output level is required, the PMOS is turned on
 - When a low output level is required, the NMOS is turned on
- Pull-up and pull-down resistors are turned on or off depending on the register status
- Read access to the output status register to get the last written value
- I/O default state is high resistance pull-down (except for special requirement I/O, such as SWD, CLOCK, etc.)

Drive Capability

Two important indicators of the device GPIO drive capability are pull current (Source) and fill current (Sink). In addition, the drive capability of a single I/O and the total I/O drive capability is also an important indicator of the chip's drive capability.

- Pull current (Source): The ability to output current externally relative to the device is the pull current capability. For example, when a 1K load resistor is connected between the device I/O and GND, the device outputs a high level and forms a loop. The process of pulling current out of the device internally by the resistor is the process of pulling current.
- Sink: Relative to the device, the ability to actively absorb current is the sink current capability. For example, when a 1K load resistor is connected between the device I/O and VCC, the device outputs a low level and forms a loop. The external power supply will pour current into the device internal process is called the process of filling current.

The device has four configurable drive capabilities per I/O, Level 0 ~ 3, where Level 0 has the worst drive capability and Level 3 has the strongest drive capability.

Function Reuse

GPIO function multiplexing is the use of general-purpose I/O for other functions, such as I²C I/O that requires SCL and SDA functions. For the device, any GPIO can be used by peripherals supported by the device, such as SPI, UART, I²C, SPI, I²S, etc. The device defines a unique ID for all peripherals, and each I/O can be set to the corresponding ID of the peripheral during initialization.

Interrupt Function

Any GPIO can generate external interrupts, and the generated external interrupts are handled by the CPU and configured into the interrupt group. The device supports external interrupts in 3 groups, GPIO_INT0 / GPIO_INT1 / GPIO_INT2 (IRQ_20, IRQ21, IRQ22) respectively. The device supports any GPIO to generate external interrupts, and the triggering method can be configured by software. The trigger methods are:

- High Level Trigger
- Low Level Trigger
- Rising edge trigger
- Falling edge trigger
- Double edge trigger

Wakeup Function

Any GPIO can wake up the chip from hibernation or sleep mode. The pin is enabled to trigger a wakeup request by configuring the corresponding register. When wakeup is enabled on the pin, the input filter (debounce function) can be enabled for debounce by configuring the corresponding register. The debounce circuit can avoid false wakeup caused by burrs. In addition, the polarity of the GPIO wakeup request can be configured.

GPIO APIs

GPIO ROM APIs

rom_hw_gpio_set_pin_cfg

Function Prototype

```
EN_ERR_STA_T rom_hw_gpio_set_pin_cfg(stGPIO_Handle_t* pstGPIO, uint32_t u32Pin,  
uint32_t u32Cfg);
```

Description

Configure GPIO pins.

Parameter

Parameter	Description
pstGPIO	GPIO peripheral handle, should be GPIOA / GPIOB / GPIOC.
u32Pin	Which pins need to be configured.
u32Cfg	Pin configuration.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_gpio_pin_init

Function Prototype

EN_ERR_STA_T rom_hw_gpio_pin_init(stGPIO_Handle_t* pstGPIO, uint32_t u32Pin, uint32_t u32Cfg);

Description

Initialize a GPIO.

Parameter

Parameter	Description
pstGPIO	GPIO peripheral handle, should be GPIOA / GPIOB / GPIOC.
u32Pin	Which pins need to be configured.
u32Cfg	Pin configuration.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_gpio_set_pin_input_output

Function Prototype

EN_ERR_STA_T rom_hw_gpio_set_pin_input_output(stGPIO_Handle_t* pstGPIO, uint32_t u32Pin, EN_GPIO_PIN_MODE_T enMode);

Description

Configure the GPIO working in input and output mode.

Parameter

Parameter	Description
pstGPIO	GPIO peripheral handle, should be GPIOA / GPIOB / GPIOC.
u32Pin	Which pins need to be configured, @ ref EN_GPIO_PIN_T.
enMode	Pin input or output mode, @ ref EN_GPIO_PIN_MODE_T.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_gpio_set_pin_output_level

Function Prototype

EN_ERR_STA_T rom_hw_gpio_set_pin_output_level(stGPIO_Handle_t* pstGPIO, uint32_t u32Pin, EN_GPIO_LEVEL_T enLevel);

Description

Set or clear the indicated pin output level.

Parameter

Parameter	Description
pstGPIO	GPIO peripheral handle, should be GPIOA / GPIOB / GPIOC.
u32Pin	Which pins need to be configured, @ ref EN_GPIO_PIN_T.
enLevel	Output level status, @ ref EN_GPIO_LEVEL_T.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_gpio_toggle_pin_output_level

Function Prototype

```
EN_ERR_STA_T rom_hw_gpio_toggle_pin_output_level(stGPIO_Handle_t* pstGPIO, uint32_t u32Pin);
```

Description

Toggle the indicated pin output level.

Parameter

Parameter	Description
pstGPIO	GPIO peripheral handle, should be GPIOA / GPIOB / GPIOC.
u32Pin	Which pins need to be configured, @ ref EN_GPIO_PIN_T.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_gpio_get_pin_output_level

Function Prototype

```
EN_ERR_STA_T rom_hw_gpio_get_pin_output_level(stGPIO_Handle_t* pstGPIO, uint32_t u32Pin, uint32_t* pu32Level);
```

Description

Get the indicated pin output status.

Parameter

Parameter	Description
pstGPIO	GPIO peripheral handle, should be GPIOA / GPIOB / GPIOC.
u32Pin	Which pins need to be configured, @ ref EN_GPIO_PIN_T.
pu32Level	Pin status, @ ref EN_GPIO_LEVEL_T.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_gpio_get_pin_input_level

Function Prototype

```
EN_ERR_STA_T rom_hw_gpio_get_pin_input_level(stGPIO_Handle_t* pstGPIO, uint32_t u32Pin, uint32_t* pu32Level);
```

Description

Get the indicated pin input status.

Parameter

Parameter	Description
pstGPIO	GPIO peripheral handle, should be GPIOA / GPIOB / GPIOC.
u32Pin	Which pins need to be configured, @ ref EN_GPIO_PIN_T.
pu32Level	Pin status, @ ref EN_GPIO_LEVEL_T.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_gpio_set_pin_pid

Function Prototype

```
EN_ERR_STA_T rom_hw_gpio_set_pin_pid(stGPIO_Handle_t* pstGPIO, uint32_t u32Pin, EN_GPIO_PID_T enPID);
```

Description

Set the indicated pin peripheral function.

Parameter

Parameter	Description
pstGPIO	GPIO peripheral handle, should be GPIOA / GPIOB / GPIOC.
u32Pin	Which pins need to be configured, @ ref EN_GPIO_PIN_T.
enPID	Pin peripheral function choice, @ ref EN_GPIO_PID_T.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_gpio_enable_qspi_pid

Function Prototype

```
EN_ERR_STA_T rom_hw_gpio_enable_qspi_pid(EN_QSPI_FIXED_GPIO_EN_T enCh);
```

Description

Enable the GPIOs for QSPI function.

Parameter

Parameter	Description
enCh	QSPI channel, @ ref EN_QSPI_FIXED_GPIO_EN_T.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_gpio_disable_qspi_pid

Function Prototype

```
EN_ERR_STA_T rom_hw_gpio_disable_qspi_pid(EN_QSPI_FIXED_GPIO_EN_T enCh);
```

Description

Disable the indicated PIN working in QSPI function.

Parameter

Parameter	Description
enCh	QSPI channel, @ ref EN_QSPI_FIXED_GPIO_EN_T.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_gpio_set_pin_pull_mode

Function Prototype

```
EN_ERR_STA_T rom_hw_gpio_set_pin_pull_mode(stGPIO_Handle_t* pstGPIO, uint32_t u32Pin, EN_GPIO_PULL_MODE_T enMode);
```

Description

Configure the indicated pin pull-up or pull-down.

Parameter

Parameter	Description
pstGPIO	GPIO peripheral handle, should be GPIOA / GPIOB / GPIOC.
u32Pin	Which pins need to be configured, @ ref EN_GPIO_PIN_T.
enMode	Pin pull mode, @ ref EN_GPIO_PULL_MODE_T.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_gpio_set_pin_drive_strength

Function Prototype

```
EN_ERR_STA_T rom_hw_gpio_set_pin_drive_strength(stGPIO_Handle_t* pstGPIO, uint32_t u32Pin, EN_GPIO_DRV_STRENGTH_T enStrength);
```

Description

Configure the indicated pin drive strength.

Parameter

Parameter	Description
pstGPIO	GPIO peripheral handle, should be GPIOA / GPIOB / GPIOC.
u32Pin	Which pins need to be configured, @ ref EN_GPIO_PIN_T.
enStrength	Pin drive strength, @ ref EN_GPIO_DRIVE_STRENGTH_T.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_gpio_set_pin_interrupt_type

Function Prototype

EN_ERR_STA_T rom_hw_gpio_set_pin_interrupt_type(stGPIO_Handle_t* pstGPIO, uint32_t u32Pin, EN_GPIO_INT_CH_T enCh, EN_GPIO_INT_TYPE_T enType);

Description

Set the indicated pin interrupt type.

Parameter

Parameter	Description
pstGPIO	GPIO peripheral handle, should be GPIOA / GPIOB / GPIOC.
u32Pin	Which pins need to be configured, @ ref EN_GPIO_PIN_T.
enCh	Interrupt handle selection, @ ref EN_GPIO_INT_CH_T.
enType	Interrupt type, @ ref EN_GPIO_INT_TYPE_T.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_gpio_get_pin_interrupt_flag

Function Prototype

EN_ERR_STA_T rom_hw_gpio_get_pin_interrupt_flag(stGPIO_Handle_t* pstGPIO, uint32_t u32Pin, uint32_t* pu32Msk);

Description

Get the indicated pin interrupt flag (status).

Parameter

Parameter	Description
pstGPIO	GPIO peripheral handle, should be GPIOA / GPIOB / GPIOC.
u32Pin	Which pins need to be configured, @ ref EN_GPIO_PIN_T.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_gpio_clear_pin_interrupt_flag

Function Prototype

```
EN_ERR_STA_T rom_hw_gpio_clear_pin_interrupt_flag(stGPIO_Handle_t* pstGPIO, uint32_t u32Pin);
```

Description

Clear the indicated pin interrupt flag (status).

Parameter

Parameter	Description
pstGPIO	GPIO peripheral handle, should be GPIOA / GPIOB / GPIOC.
u32Pin	Which pins need to be configured, @ ref EN_GPIO_PIN_T.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_gpio_enable_pin_interrupt

Function Prototype

```
EN_ERR_STA_T rom_hw_gpio_enable_pin_interrupt(stGPIO_Handle_t* pstGPIO, uint32_t u32Pin);
```

Description

Enable the indicated pin interrupt.

Parameter

Parameter	Description
pstGPIO	GPIO peripheral handle, should be GPIOA / GPIOB / GPIOC.
u32Pin	Which pins need to be configured, @ ref EN_GPIO_PIN_T.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_gpio_disable_pin_interrupt

Function Prototype

```
EN_ERR_STA_T rom_hw_gpio_disable_pin_interrupt(stGPIO_Handle_t* pstGPIO, uint32_t u32Pin);
```

Description

Disable the indicated pin interrupt.

Parameter

Parameter	Description
pstGPIO	GPIO peripheral handle, should be GPIOA / GPIOB / GPIOC.
u32Pin	Which pins need to be configured, @ ref EN_GPIO_PIN_T.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_gpio_set_pin_wakeup_debounce

Function Prototype

```
EN_ERR_STA_T rom_hw_gpio_set_pin_wakeup_debounce(uint8_t u8Time, EN_GPIO_WAKEUP_DEBOUNCE_UNIT_T enUnit);
```

Description

Set all pins wakeup debounce, all pins share one wakeup debounce time.

Parameter

Parameter	Description
u8Time	Debounce time, 0 ~ 63.
enUnit	Debounce unit, @ ref EN_GPIO_WAKEUP_DEBOUNCE_UNIT_T.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_gpio_enable_pin_wakeup

Function Prototype

```
EN_ERR_STA_T rom_hw_gpio_enable_pin_wakeup(stGPIO_Handle_t* pstGPIO, uint32_t u32Pin, EN_GPIO_WAKEUP_POL_T enPol);
```

Description

Enable the indicated pin wakeup function.

Parameter

Parameter	Description
pstGPIO	GPIO peripheral handle, should be GPIOA / GPIOB / GPIOC.
u32Pin	Which pins need to be configured, @ ref EN_GPIO_PIN_T.
enPol	Wakeup polarity, @ ref EN_GPIO_WAKEUP_POL_T.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_gpio_disable_pin_wakeup

Function Prototype

```
EN_ERR_STA_T rom_hw_gpio_disable_pin_wakeup(stGPIO_Handle_t* pstGPIO, uint32_t u32Pin);
```

Description

Disable the indicated pin wakeup function.

Parameter

Parameter	Description
pstGPIO	GPIO peripheral handle, should be GPIOA / GPIOB / GPIOC.
u32Pin	Which pins need to be configured, @ ref EN_GPIO_PIN_T.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_gpio_clear_wakeup_flag

Function Prototype

EN_ERR_STA_T rom_hw_gpio_clear_wakeup_flag(void);

Description

Clear the indicated pin wakeup flag.

Parameter

None.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

SW Port API Function

rom_hw_gpio_enable_swd

Function Prototype

EN_ERR_STA_T rom_hw_gpio_enable_swd(stGPIO_SWD_Handle_t* pstGPIO);

Description

Enable the Cortex®-M0+ or Cortex®-M33 SWD port. If the SWD port is configured to general I/O, it needs to disable the SWD port first.

Parameter

Parameter	Description
pstGPIO	GPIO peripheral handle, should be SWD_CP / SWD_MP.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_gpio_disable_swd

Function Prototype

```
EN_ERR_STA_T rom_hw_gpio_disable_swd(stGPIO_SWD_Handle_t* pstGPIO);
```

Description

Disable the Cortex®-M0+ or Cortex®-M33 SWD port. If the SWD port is configured to general I/O, it needs to disable the SWD port first.

Parameter

Parameter	Description
pstGPIO	GPIO peripheral handle, should be SWD_CP / SWD_MP.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_gpio_set_sw_input_output

Function Prototype

```
EN_ERR_STA_T rom_hw_gpio_set_sw_input_output(stGPIO_SWD_Handle_t* pstGPIO, uint8_t u8Pin, EN_GPIO_PIN_MODE_T enMode);
```

Description

Configure the SWD interface pin (SWDIO or SWCLK) input or output mode.

Parameter

Parameter	Description
pstGPIO	GPIO peripheral handle, should be SWD_CP / SWD_MP.
u8Pin	The pins which would be configured, @ ref EN_GPIO_SW_NUM_T.
enMode	Pin input or output mode, @ ref EN_GPIO_PIN_MODE_T.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_gpio_set_sw_output_level

Function Prototype

```
EN_ERR_STA_T rom_hw_gpio_set_sw_output_level(stGPIO_SWD_Handle_t* pstGPIO, uint8_t u8Pin, EN_GPIO_LEVEL_T enLevel);
```

Description

Set the indicated pin (SWDIO or SWCLK) output status.

Parameter

Parameter	Description
pstGPIO	GPIO peripheral handle, should be SWD_CP / SWD_MP.
u8Pin	The pins which would be configured, @ ref EN_GPIO_SW_NUM_T.
enLevel	Output level status, @ ref EN_GPIO_LEVEL_T.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_gpio_get_sw_input_level

Function Prototype

EN_ERR_STA_T rom_hw_gpio_get_sw_input_level(stGPIO_SWD_Handle_t* pstGPIO, uint8_t u8Pin, uint8_t* pu8Level);

Description

Get the indicated pin (SWDIO or SWCLK) input status.

Parameter

Parameter	Description
pstGPIO	GPIO peripheral handle, should be SWD_CP / SWD_MP.
u8Pin	The pins which would be configured, @ ref EN_GPIO_SW_NUM_T.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_gpio_set_sw_pull_mode

Function Prototype

EN_ERR_STA_T rom_hw_gpio_set_sw_pull_mode(stGPIO_SWD_Handle_t* pstGPIO, uint8_t u8Pin, EN_GPIO_PULL_MODE_T enMode);

Description

Configure the SWD interface pin (SWDIO or SWCLK) pull-up mode or pull-down mode.

Parameter

Parameter	Description
pstGPIO	GPIO peripheral handle, should be SWD_CP / SWD_MP.
u8Pin	The pins which would be configured, @ ref EN_GPIO_SW_NUM_T.
enMode	Pin pull mode, @ ref EN_GPIO_PULL_MODE_T.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_gpio_set_sw_drive_strength

Function Prototype

EN_ERR_STA_T rom_hw_gpio_set_sw_drive_strength(stGPIO_SWD_Handle_t* pstGPIO, uint8_t u8Pin, EN_GPIO_DRV_STRENGTH_T enStrength);

Description

Configure the SWD interface pin (SWDIO or SWCLK) drive strength.

Parameter

Parameter	Description
pstGPIO	GPIO peripheral handle, should be SWD_CP / SWD_MP.
u8Pin	The pins which would be configured, @ ref EN_GPIO_SW_NUM_T.
enStrength	Pin drive strength, @ ref EN_GPIO_DRV_STRENGTH_T.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

IR API Function

rom_hw_gpio_enable_ir_tx_out

Function Prototype

```
EN_ERR_STA_T rom_hw_gpio_enable_ir_tx_out(void);
```

Description

Enable the IR TX output.

Parameter

None.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_gpio_disable_ir_tx_out

Function Prototype

```
EN_ERR_STA_T rom_hw_gpio_disable_ir_tx_out(void);
```

Description

Disable the IR TX output.

Parameter

None.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_gpio_enable_ir_rx_amp

Function Prototype

```
EN_ERR_STA_T rom_hw_gpio_enable_ir_rx_amp(void);
```

Description

Enable the IR RX AMP.

Parameter

None.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_gpio_disable_ir_rx_amp

Function Prototype

```
EN_ERR_STA_T rom_hw_gpio_disable_ir_rx_amp(void);
```

Description

Disable the IR RX AMP.

Parameter

None.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_gpio_set_ir_rx_rtune

Function Prototype

```
EN_ERR_STA_T rom_hw_gpio_set_ir_rx_rtune(uint8_t u8Rtune);
```

Description

Set the resistance of R in RC filter.

Parameter

Parameter	Description
u8Rtune	$R = 50K \times 2^{(u8Rtune)}$.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_gpio_get_ir_ctrl_value

Function Prototype

```
EN_ERR_STA_T rom_hw_gpio_get_ir_ctrl_value(uint32_t* pu32IrCtrlValue);
```

Description

Get the IR control register value.

Parameter

Parameter	Description
pu32IrCtrlValue	Pointer to a 32-bit buffer used to return the IR CTRL register value.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

DCXO Pins API Function

rom_hw_gpio_enable_dcxo32k_output

Function Prototype

EN_ERR_STA_T rom_hw_gpio_enable_dcxo32k_output(uint8_t u8Io);

Description

Enable the DCXO32K clock output from PB07 (P39) or (and) PB10 (P42).

Parameter

Parameter	Description
u8Io	Output I/O, @ ref EN_DCXO32K_OUT_CFG_T.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_gpio_disable_dcxo32k_output

Function Prototype

EN_ERR_STA_T rom_hw_gpio_disable_dcxo32k_output(uint8_t u8Io);

Description

Disable the DCXO32K clock output from PB07 (P39) or (and) PB10 (P42).

Parameter

Parameter	Description
u8Io	Output I/O, @ ref EN_DCXO32K_OUT_CFG_T.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

GPIO Examples

The main functions of GPIO have been elaborated in the previous section, and this section will develop the DEMO project based on the main functions of GPIO.

Example Code for Input

The input project is mainly used to test whether the GPIO can read the externally transmitted signal through the I/O port. The sample code is as follows.

```
uint32_t i;  
uint32_t u32Level = 0;  
rom_hw_gpio_set_pin_pull_mode(GPIO_PORT_INPUT_IO, GPIO_PIN_INPUT_IO,  
    enPullMode);  
rom_hw_gpio_set_pin_input_output(GPIO_PORT_INPUT_IO, GPIO_PIN_INPUT_IO,  
    GPIO_MODE_INPUT);  
for (i = 0; i < 16; i++)  
{  
    rom_hw_gpio_get_pin_input_level(GPIO_PORT_INPUT_IO, GPIO_PIN_INPUT_IO, &u  
        32Level);  
    PRINTF ("GPIO input level: %08X\n", u32Level);  
    rom_delay_ms (1000);  
}
```

Example Code for Output

The output sample project is mainly to test whether the GPIO can transmit signals outward through the I/O port. The sample code is as follows.

```
uint32_t u32Level;  
rom_hw_gpio_set_pin_input_output(GPIO_PORT_OUTPUT_IO, GPIO_PIN_OUTPUT_IO,  
    GPIO_MODE_OUTPUT);  
rom_hw_gpio_set_pin_drive_strength(GPIO_PORT_OUTPUT_IO, GPIO_PIN_OUTPUT_IO,  
    enStrength);  
rom_hw_gpio_set_pin_output_level(GPIO_PORT_OUTPUT_IO, GPIO_PIN_OUTPUT_IO, 1);  
rom_hw_gpio_get_pin_output_level(GPIO_PORT_OUTPUT_IO, GPIO_PIN_OUTPUT_IO,  
    &u32Level);  
PRINTF("(1) GPIO output level: %08X\n", u32Level);  
rom_delay_ms (1000);  
rom_hw_gpio_set_pin_output_level(GPIO_PORT_OUTPUT_IO, GPIO_PIN_OUTPUT_IO, 0);  
rom_hw_gpio_get_pin_output_level(GPIO_PORT_OUTPUT_IO, GPIO_PIN_OUTPUT_IO,  
    &u32Level);  
PRINTF("(2) GPIO output level: %08X\n", u32Level);
```

Example Code for GPIO Interrupt

This sample project is mainly used to test whether it can respond to external interrupts generated via GPIO. The example project is as follows.

Interrupt Handler

```
uint32_t u32Flag, u32Level;  
/* Get interrupt status */  
rom_hw_gpio_get_pin_interrupt_flag(GPIO_PORT_INPUT_IO, GPIO_PIN_INPUT_IO,  
    &u32Flag);  
rom_hw_gpio_clear_pin_interrupt_flag(GPIO_PORT_INPUT_IO, GPIO_PIN_INPUT_IO);  
PRINTF("io int flag: %08X\n", u32Flag);  
rom_hw_gpio_get_pin_input_level(GPIO_PORT_INPUT_IO, GPIO_PIN_INPUT_IO, &u  
    32Level);  
PRINTF("io int level: %08X\n", u32Level);
```

Interrupt DEMO

```
rom_hw_gpio_set_pin_input_output(GPIO_PORT_INPUT_IO, GPIO_PIN_INPUT_IO, GPIO_
MODE_INPUT);
if (GPIO_INT_HIGH_LEVEL == enType)
{
    rom_hw_gpio_set_pin_pull_mode(GPIO_PORT_INPUT_IO, GPIO_PIN_INPUT_IO, G PIO_
        PULL_DOWN);
}
else
{
    rom_hw_gpio_set_pin_pull_mode(GPIO_PORT_INPUT_IO, GPIO_PIN_INPUT_IO, G PIO_
        PULL_UP);
}
/* Configure for gpio interrupt */
/* Registration interruption */
rom_hw_sys_ctrl_enable_peri_int (SYS_CTRL_MP, GPIO_IRQ0);
/* Enable gpio1 interrupt */
NVIC_ClearPendingIRQ (GPIO_IRQ0);
NVIC_SetPriority (GPIO_IRQ0, 0x3);
NVIC_EnableIRQ (GPIO_IRQ0);
/* Disable gpio interrupt */
rom_hw_gpio_disable_pin_interrupt (GPIO_PORT_INPUT_IO, GPIO_PIN_INPUT_IO);
/* Initialize io to interrupt mode. */
rom_hw_gpio_set_pin_interrupt_type (GPIO_PORT_INPUT_IO, GPIO_PIN_INPUT_IO,
GPIO_INT_CH0, enType);
/* Enable gpio interrupt */
rom_hw_gpio_enable_pin_interrupt (GPIO_PORT_INPUT_IO, GPIO_PIN_INPUT_IO);
```

Example Code for GPIO Wakeup

This sample project is to test whether it is possible to wake up a CPU in sleep via GPIO.

```
#define WAKEUP_PIN GPIO_21
/* Configure wakeup source */
rom_hw_pmu_set_gpio_wakeup_source(n, PMU_LUT_TRIG_GPIOB, WAKEUP_PIN, LUT_ACT_PD_
SYS_ON);
/* Configure debounce time */
rom_hw_gpio_set_pin_wakeup_debounce(60, GPIO_WAKEUP_DEB_UNIT_30US);
/* enable wakeup */
rom_hw_gpio_enable_pin_wakeup(GPIOB, WAKEUP_PIN, GPIO_WAKEUP_HIGH_LEVEL);
```

3 General-Purpose Timer (GPTM)

Introduction

The General Purpose Timer, GPTM, also known as GTIM, is a 32-bit timer configured with a 4-channel comparator that enables timer functions. The GPTM operates in the 32K clock domain, and after the device enters sleep, the GPTM can still operate and wake up the system. The GPTM introduces a flexible clocking scheme that provides the required functionality and performance while minimizing power consumption.

Features

- 16 MHz / 32 kHz clock or GPIO trigger source can be used as clock source
- Interrupt type: cpu_cap / overflow / matching / capture_coverd / decode
- 2-way PWM and combined 32-bit PWM
- 1 way infrared emission
- 2-way GPIO / IR trigger capture function
 - Support CNT_A / CNT_B for capture (not support simultaneous)
 - Supports combined 32-bit for capture (CNT_A enabled only)
- 2-way GPIO / IR trigger decode function
 - Support CNT_A / CNT_B for decode (support simultaneous)
 - Supports combined 32-bit for decode (CNT_A enabled only)
- Supports decode-capture loopback (CNT_A does decode, CNT_B does capture)
- Can be used as one 32-bit counter or two 16-bit divider counters
- Support interrupt capture mode (only one can be selected at the same time as DMA capture mode)
- Support capture level indication (in 16-bit mode, capcnt[15] indicates CNT_B high and low level (high), capcnt[31] indicates CNT_A level, in 32-bit mode, capcnt[31] indicates 32-bit combined cnt level)
- Support multiple GPTM start and stop at the same time
- The compare value supports the next cycle in effect

Functional Description

The device mainly has the functions of counting, generating PWM waves and sampling.

Counter Mode (CNT Mode)

When the GPTM works in the counter mode, the 32-bit counter register contains two 16-bit counters, CNT_A and CNT_B see below:

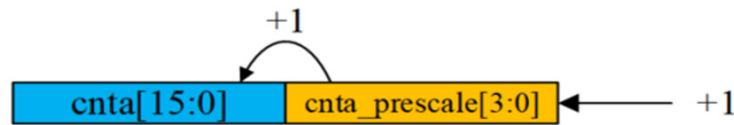


Figure 3. GPTM Counter Register

As shown above, when the CNT_A count input is valid, cnta_prescale counts first and cnta adds 1 when the prescale count reaches the set value.

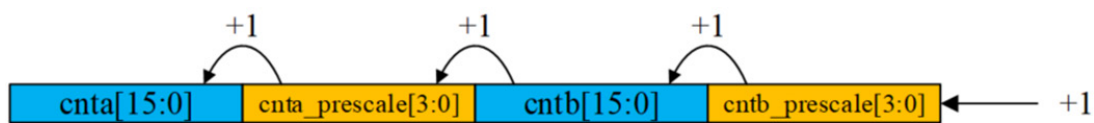


Figure 4. GPTM Prescaler

When two cnts are used in combination, the cnta_prescale value should be set to 0. cntb_prescale counts first, and when the count reaches the set value, 1 is added to CNT_B. When CNT_B is full, it is fed into cnta_prescale, and after cnta_prescale counts to the set value, it is fed into CNT_A.

The count starts in incremental mode, and the register controls whether the change point is a compare point (a compare value greater than zero is considered a valid value) or an overflow point, and whether the change after reaching the change point is reset (reset to 0 for incremental; reset to full f for decremental) or gradual (change to decremental for incremental; change to incremental for decremental). The principle of compare value setting is shown in the table below.

Table 3. GPTM Mode Selection

rst_mode	rstval_mode	Increase	Decrease
0	0	Counting to compare continues counting and resets to 0 when it reaches 16'hffff_ffff.	Counting to compare continues counting and resets to 16'hffff_ffff when it reaches 0.
0	1	The count is reset to 0 when it reaches compare, and the count continues to increment.	The count is reset to 16'hffff_ffff when it reaches compare, and the count continues to decrement.
1	0	Counting to compare continues counting and becomes decreasing when it reaches 16'hffff_ffff.	Counting to compare continues counting and becomes incremental when it reaches 0.
1	1	The count to compare becomes decreasing.	The count to compare becomes incremental.

The above table shows the working principle in different modes. In order to show the principle more graphically, the timing diagram in different modes will be shown below.

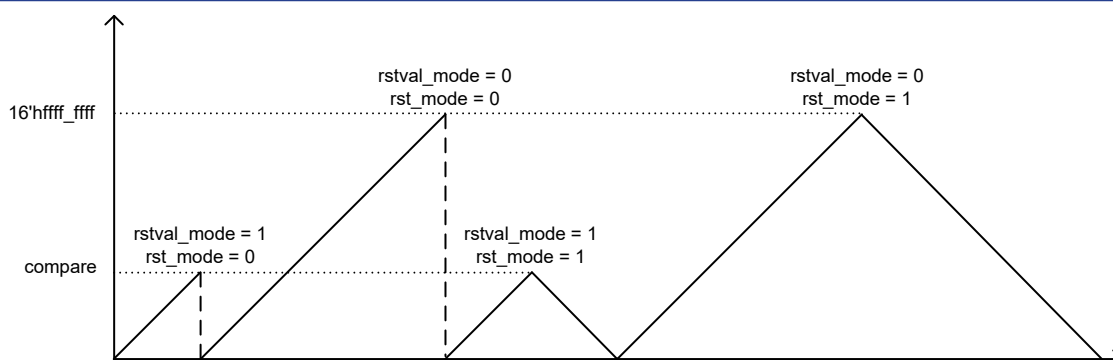
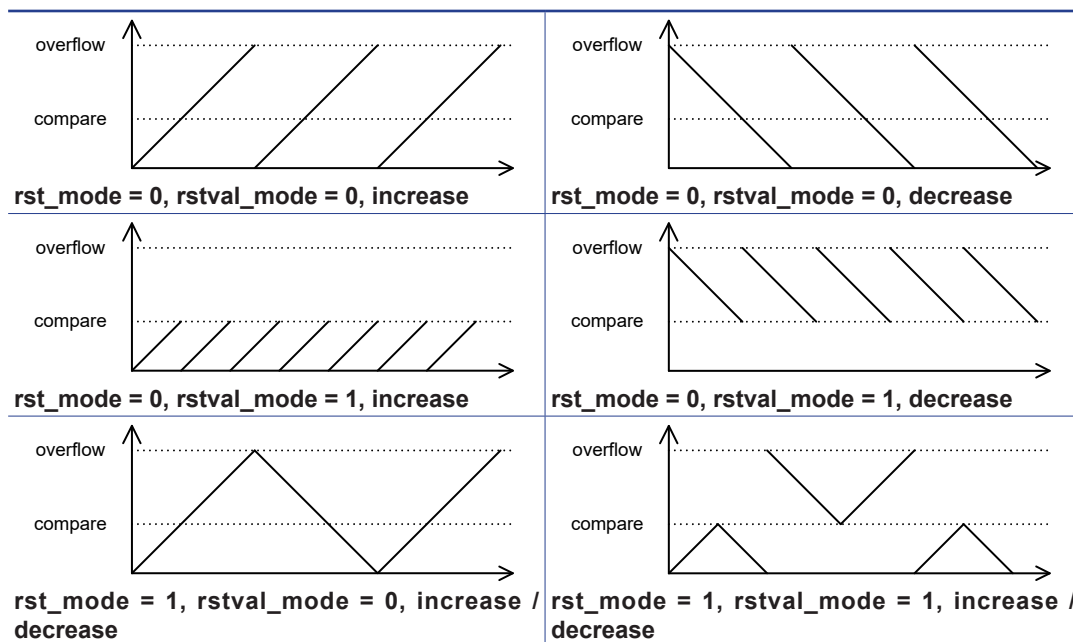


Figure 5. Timing of GPTM Work Mode

General Functional Timing Description

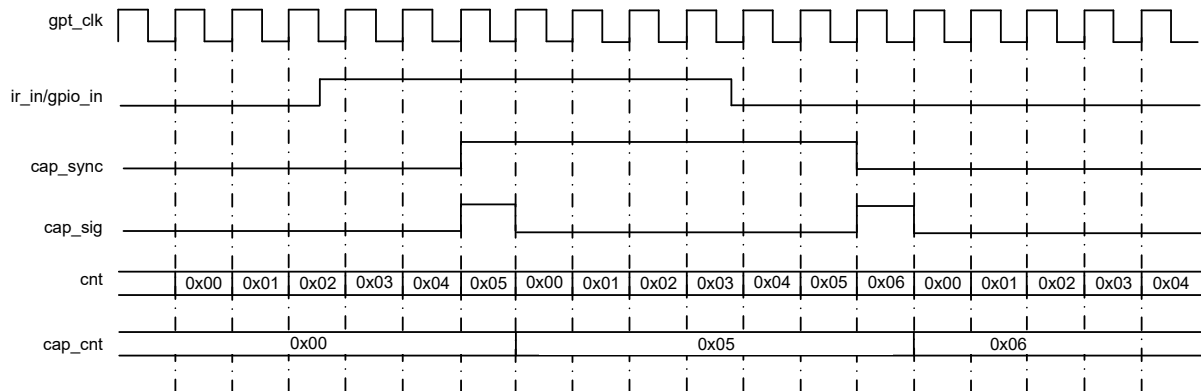


Capture Mode

The capture function is used to detect the edge interval of the IR input signal or GPIO input signal.

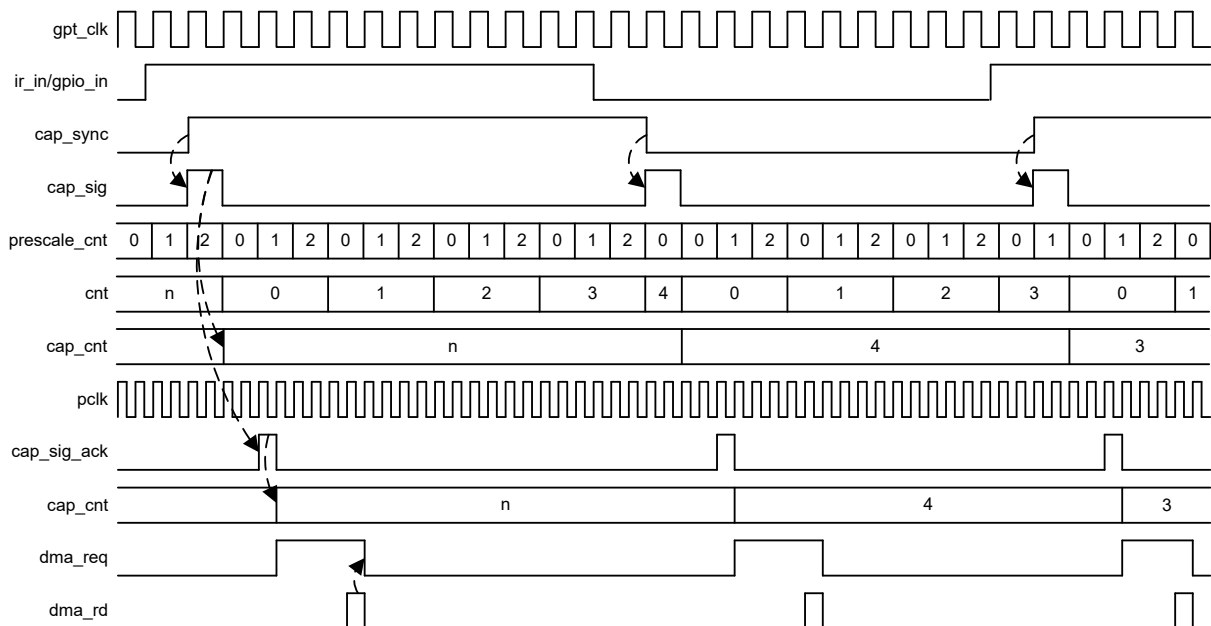
After synchronizing the input signal, the edge detection is performed, and rising edge, falling edge or double edge sampling can be selected. For double edge sampling, the highest bit can be used as the level indication using the level indication function. Each sampling point will first save the cnt value to cnt_tmp, and zero the cnt value, and generate a dma request signal to request dma to carry data, or use interrupt capture mode to generate an interrupt, and the CPU will read the cnt data.

If a new sample point arrives during the time period between request generation and data handling, the sample point is ignored and the capcov interrupt is reported. Note that the sample point will clear cnt, but will not overwrite the cnt_tmp value. The basic functions of capture are shown in the figure below.



From the above figure, we can see that in the case of cnt without dividing frequency, since the action of clearing zero needs to occupy one cycle or the signal edge itself will occupy one cycle, so the actual counting of two signal edges is from 0x0 to 0x6 for a total of 7 cycles, at this time, we can directly use cap_cnt plus 1 as the actual length.

When there is a prescale frequency, such as 3 division, as shown below, due to the error caused by the division itself, cap_sig may arrive at 2, 0 or 1 of the division counter prescale_cnt, then the actual counting period (gpt_clk) should be $\text{cap_cnt} \times 3 + 3$, $\text{cap_cnt} \times 3 + 1$, $\text{cap_cnt} \times 3 + 2$ respectively, and at this time, the direct cap_cnt value plus 1 cannot be used as the actual cnt count.



Capture new plus level indication function: Since it is difficult to distinguish the upper and lower edges by double edge sampling, it occupies 1-bit highest bit as level indication (please turn off this function for non-double edge sampling), bit[15] as level indication bit when 16-bit cnt counting, and bit[31], as level indication when 32-bit cnt counting, 1 means high level. When using this function, the compare should be configured as 15'h7fff (16-bit cnt) or 31'h7fff_ffff as overflow interrupt because the highest bit is occupied.

PWM Mode

The GPTM module is also used to generate the specified waveform. The working principle is described below.

The compare value is the count cycle, and the specified level is output when the count value is between pwm_low and pwm_hi (the high and low levels are determined by pwm_pol).

The values of pwm_low and pwm_hi take effect for the first time after the start of the configuration completion trigger, and later after the end of the compare cycle. Compare can take effect in the current cycle or in the next cycle.

Note: $\text{pwm_low} \leq \text{cnt} < \text{pwm_hi}$ outputs the specified level, so configure duty cycle to 0%: pwm_low and pwm_hi are both greater than compare or $\text{pwm_low} \geq \text{pwm_hi}$; configure duty cycle to 100%: $\text{pwm_low} = 0$, pwm_hi is greater than compare (compare is configured to full f, 100% duty cycle cannot be generated in this way).

GPTM APIs

Timer Function APIs

rom_hw_timer_trig_cfg_valid

Function Prototype

```
EN_ERR_STA_T rom_hw_timer_trig_cfg_valid(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh);
```

Description

Make the timer configuration valid.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which channel will be configured, refer to the EN_TIMER_CH_T enumeration definition.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_timer_set_work_mode

Function Prototype

```
EN_ERR_STA_T rom_hw_timer_set_work_mode(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh);
```

Description

Select the timer work mode by writing the TIMER_MODE register with indicated bits.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which channel will be configured, refer to the EN_TIMER_CH_T enumeration definition.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_timer_get_mode

Function Prototype

```
EN_ERR_STA_T rom_hw_timer_get_mode(stTIMER_Handle_t* pstTIMER, uint32_t* pu32Mode);
```

Description

Get the timer work mode.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
pu32Mode	Point to timer mode configuration.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_timer_set_clock_prescale

Function Prototype

```
EN_ERR_STA_T rom_hw_timer_set_clock_prescale(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, uint8_t u8Prescale);
```

Description

Configure the indicated timer clock prescale.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which channel will be configured, refer to the EN_TIMER_CH_T enumeration definition.
u8Prescale	The divider for the input clock, the range of u8Prescale is 0 to 0x0F.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_timer_set_counter_mode

Function Prototype

```
EN_ERR_STA_T rom_hw_timer_set_counter_mode(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, EN_TIMER_CNT_MODE_T enMode);
```

Description

Configure the indicated timer to work in counter mode.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which channel will be configured, refer to the EN_TIMER_CH_T enumeration definition.
enMode	Timer counter mode control, refer to the EN_TIMER_CNT_MODE_T enumeration definition.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_timer_get_cfg

Function Prototype

EN_ERR_STA_T rom_hw_timer_get_cfg(stTIMER_Handle_t* pstTIMER, uint32_t* pu32Cfg);

Description

Get the timer configuration.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
pu32Cfg	Point to timer configuration.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_timer_start

Function Prototype

EN_ERR_STA_T rom_hw_timer_start(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh);

Description

Start the indicated timer.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which channel will be configured, refer to the EN_TIMER_CH_T enumeration definition.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_timer_stop

Function Prototype

```
EN_ERR_STA_T rom_hw_timer_stop(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh);
```

Description

Stop the indicated timer.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which channel will be configured, refer to the EN_TIMER_CH_T enumeration definition.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_timer_get_counter

Function Prototype

```
EN_ERR_STA_T rom_hw_timer_get_counter(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, uint32_t* pu32Cnt);
```

Description

Get the indicated timer current counter.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which channel will be configured, refer to the EN_TIMER_CH_T enumeration definition.
pu32Cnt	Current timer counter.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_timer_clear_counter

Function Prototype

```
EN_ERR_STA_T rom_hw_timer_clear_counter(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh);
```

Description

Clear the indicated timer counter.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which channel will be configured, refer to the EN_TIMER_CH_T enumeration definition.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_timer_set_compare

Function Prototype

```
EN_ERR_STA_T rom_hw_timer_set_compare(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, uint32_t u32Value);
```

Description

Configure the indicated timer compare value.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which channel will be configured, refer to the EN_TIMER_CH_T enumeration definition.
u32Value	Compare value. 16-bit timer: the range of compare value is 0 to 0xFFFF. 32-bit timer: the range of compare value is 0 to 0xFFFFFFFF.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_timer_get_compare

Function Prototype

```
EN_ERR_STA_T rom_hw_timer_get_compare(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, uint32_t* pu32Value);
```

Description

Get the indicated timer compare value.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which channel will be configured, refer to the EN_TIMER_CH_T enumeration definition.
pu32Value	Point to compare value.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_timer_get_interrupt_flag

Function Prototype

```
EN_ERR_STA_T rom_hw_timer_get_interrupt_flag(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, uint32_t* pu32Msk);
```

Description

Get the indicted timer interrupt flags (status) by reading the TIMER_INT_FLAG register.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which channel will be configured, refer to the EN_TIMER_CH_T enumeration definition.
pu32Msk	Indicate which interrupt flag will be read.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_timer_clear_interrupt_flag

Function Prototype

```
EN_ERR_STA_T rom_hw_timer_clear_interrupt_flag(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, uint32_t u32Msk);
```

Description

Clear the indicated TIMER interrupt flag (status) by writing the TIMER_INT_CLR register.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which channel will be configured, refer to the EN_TIMER_CH_T enumeration definition.
pu32Msk	Indicate which flag will be cleared.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_timer_enable_interrupt

Function Prototype

```
EN_ERR_STA_T rom_hw_timer_enable_interrupt(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, uint32_t u32IntEn);
```

Description

Enable the indicated TIMER interrupt by writing the TIMER_INT_EN register with indicated bits.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which channel will be configured, refer to the EN_TIMER_CH_T enumeration definition.
u32IntEn	Indicate which interrupt will be enabled. Bit = 1 means enable Bit = 0 means no impact Refer to the EN_TIMER_INT_FLAG_T enumeration definition.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_timer_disable_interrupt

Function Prototype

EN_ERR_STA_T rom_hw_timer_disable_interrupt(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, uint32_t u32IntDis);

Description

Disable the indicated TIMER interrupt by writing the TIMER_INT_EN register with indicated bits.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which channel will be configured, refer to the EN_TIMER_CH_T enumeration definition.
u32IntDis	Indicate which interrupt will be disabled Bit = 1 means disable Bit = 0 means no impact Refer to the EN_TIMER_INT_FLAG_T enumeration definition.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_timer_enable_wakeup

Function Prototype

EN_ERR_STA_T rom_hw_timer_enable_wakeup(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, uint32_t u32En);

Description

Enable the timer wakeup function.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which channel will be configured, refer to the EN_TIMER_CH_T enumeration definition.

Parameter	Description
u32En	Indicate which wakeup source will be enabled, refer to the EN_TIMER_WAKEUP_SRC_T enumeration definition.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_timer_disable_wakeup

Function Prototype

```
EN_ERR_STA_T rom_hw_timer_disable_wakeup(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, uint32_t u32Dis);
```

Description

Disable timer wakeup function.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which channel will be configured, refer to the EN_TIMER_CH_T enumeration definition.
u32Dis	Indicate which wakeup source will be disabled, refer to the EN_TIMER_WAKEUP_SRC_T enumeration definition.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_timer_enable_sync_start

Function Prototype

```
EN_ERR_STA_T rom_hw_timer_enable_sync_start(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh);
```

Description

Enable the indicated timer start to work synchronously function.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which channel will be configured, refer to the EN_TIMER_CH_T enumeration definition.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_timer_disable_sync_start

Function Prototype

```
EN_ERR_STA_T rom_hw_timer_disable_sync_start(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh);
```

Description

Disable the indicated timer start to work synchronously function.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which channel will be configured, refer to the EN_TIMER_CH_T enumeration definition.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_timer_enable_sync_stop

Function Prototype

```
EN_ERR_STA_T rom_hw_timer_enable_sync_stop(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh);
```

Description

Enable the indicated timer stop working synchronously function.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which channel will be configured, refer to the EN_TIMER_CH_T enumeration definition.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_timer_disable_sync_stop

Function Prototype

```
EN_ERR_STA_T rom_hw_timer_disable_sync_stop(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh);
```

Description

Disable the indicated timer stop working synchronously function.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which channel will be configured, refer to the EN_TIMER_CH_T enumeration definition.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_timer_enable_compare_valid_delay

Function Prototype

```
EN_ERR_STA_T rom_hw_timer_enable_compare_valid_delay(stTIMER_Handle_t* pstTIMER,
EN_TIMER_CH_T enCh);
```

Description

Configure the indicated timer compare value valid for the next compare period.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which timer will be configured, refer to the EN_TIMER_CH_T enumeration definition.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_timer_disable_compare_valid_delay

Function Prototype

```
EN_ERR_STA_T rom_hw_timer_disable_compare_valid_delay(stTIMER_Handle_t* pstTIMER,
EN_TIMER_CH_T enCh);
```

Description

Configure the indicated timer compare value valid immediately.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which timer will be configured, refer to the EN_TIMER_CH_T enumeration definition.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_timer_get_work_status

Function Prototype

```
EN_ERR_STA_T rom_hw_timer_get_work_status(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh,
uint8_t* pu8Status);
```

Description

Get the indicated timer work status.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which timer, refer to the EN_TIMER_CH_T enumeration definition.
pu8Status	Timer work status, refer to the EN_TIMER_STATUS_T enumeration definition. TIMER_STATUS_RUNNING: Timer is running; TIMER_STATUS_STOP: Timer is disabled or stopped.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_timer_get_counter_mode

Function Prototype

EN_ERR_STA_T rom_hw_timer_get_counter_mode(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, uint8_t* pu8Mode);

Description

Get the indicated timer counter mode.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	enCh: Indicate which timer, refer to the EN_TIMER_CH_T enumeration definition.
pu8Mode	Timer counter mode, refer to the EN_TIMER_COUNTER_MODE_T enumeration definition. TIMER_COUNTER_MODE_DECREASE: Timer is working at decrease mode; TIMER_COUNTER_MODE_INCREASE: Timer is working at increase mode.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_timer_get_wakeup_status

Function Prototype

EN_ERR_STA_T rom_hw_timer_get_wakeup_status(stTIMER_Handle_t* pstTIMER, uint8_t* pu8State);

Description

Get timer wakeup source work status.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
pu8Status	Point to save timer wakeup source work status.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hal_timer_enable_clk

Function Prototype

EN_ERR_STA_T rom_hal_timer_enable_clk(stTIMER_Handle_t* pstTIMER);

Description

Enable timer CLK.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hal_timer_disable_clk

Function Prototype

EN_ERR_STA_T rom_hal_timer_disable_clk(stTIMER_Handle_t* pstTIMER);

Description

Disable timer clk.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hal_timer_init

Function Prototype

EN_ERR_STA_T rom_hal_timer_init(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, stTimerInit_t* pstTimerInit);

Description

Initialize the indicated timer.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which channel will be enabled, refer to the EN_TIMER_CH_T enumeration definition.
pstTimerInit	Initialize parameters of timer, refer to the stTimerInit_t enumeration definition.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hal_timer_set_compare

Function Prototype

```
EN_ERR_STA_T rom_hal_timer_set_compare(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, uint32_t u32Value);
```

Description

Configure the indicated timer compare value.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which channel will be configured with compare value.
u32Value	Compare value. 16-bit timer: the range of compare value is 0 to 0xFFFF. 32-bit timer: the range of compare value is 0 to 0xFFFFFFFF.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

PWM Function APIs

rom_hw_timer_enable_pwm

Function Prototype

```
EN_ERR_STA_T rom_hw_timer_enable_pwm(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh);
```

Description

Enable the PWM module.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which timer PWM will be enabled, refer to the EN_TIMER_CH_T enumeration definition.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_timer_disable_pwm

Function Prototype

```
EN_ERR_STA_T rom_hw_timer_disable_pwm(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh);
```

Description

Disable the PWM module.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which timer PWM will be disabled, refer to the EN_TIMER_CH_T enumeration definition.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_timer_set_pwm_hi_low_cnt

Function Prototype

EN_ERR_STA_T rom_hw_timer_set_pwm_hi_low_cnt(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, uint32_t u32HiCount, uint32_t u32LoCount);

Description

Configure the indicated timer compare high and low counter.

Note: 16-bit timer: the range of counter value is 0 to 0xFFFF.

32-bit timer: the range of counter value is 0 to 0xFFFFFFFF.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which channel will be configured.
u32HiCount	Compare high counter.
u32LoCount	Compare low counter.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_timer_get_pwm_hi_low_cnt

Function Prototype

EN_ERR_STA_T rom_hw_timer_get_pwm_hi_low_cnt(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, uint32_t* pu32HiCount, uint32_t* pu32LoCount);

Description

Get the specified PWM channel of high and low counter.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which channel will be configured.
pu32HiCount	Point to PWM high counter.
pu32LoCount	Point to PWM low counter.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_timer_set_pwm_polarity

Function Prototype

EN_ERR_STA_T rom_hw_timer_set_pwm_polarity(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, EN_PWM_POL_T enPol);

Description

Set the PWM output polarity.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which channel will be configured.
enPol	PWM polarity, refer to the EN_PWM_POLARITY_T enumeration definition.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hal_timer_pwm_init

Function Prototype

EN_ERR_STA_T rom_hal_timer_pwm_init(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, uint32_t u32Frequency, uint16_t u16Duty);

Description

Configure the indicated PWM frequency and duty.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which PWM will be configured with frequency and duty, refer to the EN_PWM_CH_T enumeration definition.
u32Frequency	PWM output frequency, unit: Hz
u16Duty	PWM duty: 0 ~ 10000 → (0 % ~ 100.00 %), accuracy: Timer CLK / PWM Frequency

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hal_timer_pwm_hi_lo_cnt

Function Prototype

```
EN_ERR_STA_T rom_hal_timer_set_pwm_hi_lo_cnt(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, uint32_t u32HiCount, uint32_t u32LoCount);
```

Description

Configure the indicated PWM compare value and high, low counter.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which PWM will be configured, refer to the EN_PWM_CH_T enumeration definition.
u32HiCount	PWM high counter
u32LoCount	PWM low counter

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hal_timer_enable_pwm

Function Prototype

```
EN_ERR_STA_T rom_hal_timer_enable_pwm(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh);
```

Description

Enable the PWM module.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which PWM will be enabled, refer to the EN_PWM_CH_T enumeration definition.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hal_timer_disable_pwm

Function Prototype

```
EN_ERR_STA_T rom_hal_timer_disable_pwm(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh);
```

Description

Disable the PWM module.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which PWM will be disabled, refer to the EN_PWM_CH_T enumeration definition.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

Timer Capture Function

rom_hw_timer_enable_capture

Function Prototype

EN_ERR_STA_T rom_hw_timer_enable_capture(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh);

Description

Enable the timer capture function.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which channel will be enabled, refer to the EN_PWM_CH_T enumeration definition.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_timer_disable_capture

Function Prototype

EN_ERR_STA_T rom_hw_timer_disable_capture(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh);

Description

Disable the timer capture function.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which channel will be disabled, refer to the EN_PWM_CH_T enumeration definition.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_timer_set_capture_and_decode_src

Function Prototype

```
EN_ERR_STA_T rom_hw_timer_set_capture_and_decode_src(stTIMER_Handle_t* pstTIMER,
EN_TIMER_CH_T enCh, EN_CAP_DECODE_SIGNAL_T enSignal);
```

Description

Configure the capture and decode signal source.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which channel, refer to the EN_PWM_CH_T enumeration definition.
enSignal	Capture signal, refer to the EN_CAP_DECODE_SIGNAL_T enumeration definition.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_timer_set_capture_chb_src

Function Prototype

```
EN_ERR_STA_T rom_hw_timer_set_capture_chb_src(stTIMER_Handle_t* pstTIMER, EN_CAP_CHB_SRC_T enSrc);
```

Description

Set the indicated timer capture channel B signal source.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enSrc	Capture channel B signal source, refer to the EN_CAP_CHB_SRC_T enumeration definition. TIMER_CAP_CHB_SRC_GPIO_IR: Capture channel B signal source is GPIO or IR TIMER_CAP_CHB_SRC_DECODE: Capture channel B signal source is channel A decode signal

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_timer_set_capture_mode

Function Prototype

```
EN_ERR_STA_T rom_hw_timer_set_capture_mode(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, EN_CAP_MODE_T enMode);
```

Description

Configure the capture mode.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which timer will be configured, refer to the EN_TIMER_CH_T enumeration definition.
enMode	Capture mode, refer to the EN_CAP_MODE_T enumeration definition.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_timer_get_capture_counter

Function Prototype

EN_ERR_STA_T rom_hw_timer_get_capture_counter(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, uint32_t* pu32CapCnt);

Description

Get the indicated timer capture counter.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which channel, refer to the EN_TIMER_CH_T enumeration definition.
pu32CapCnt	Point to capture counter.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_timer_set_capture_trig_mode

Function Prototype

EN_ERR_STA_T rom_hw_timer_set_capture_trig_mode(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, EN_CAP_TRIG_MODE_T enMode);

Description

Configure the capture trigger mode.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which channel, refer to the EN_TIMER_CH_T enumeration definition.
enMode	Capture trigger mode, refer to the EN_CAP_TRIG_MODE_T enumeration definition.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_timer_set_capture_counter_format

Function Prototype

```
EN_ERR_STA_T rom_hw_timer_set_capture_counter_format(stTIMER_Handle_t* pstTIMER,
EN_TIMER_CH_T enCh, EN_CAP_CNT_FORMAT_T enFormat);
```

Description

Configure the capture counter format.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which channel, refer to the EN_TIMER_CH_T enumeration definition.
enFormat	Capture counter format, refer to the EN_CAP_CNT_FORMAT_T enumeration definition. CAP_CNT_FORMAT_UNSIGNED: Capture counter is unsigned. CAP_CNT_FORMAT_SIGNED: Capture counter is signed. When capture work at 16-bit, counter[15] = 0, means input signal is low level; counter[15] = 1, means input signal is high level. When capture work at 32-bit, counter[31] = 0, means input signal is low level; counter[31] = 1, means input signal is high level.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hal_timer_capture_init

Function Prototype

```
EN_ERR_STA_T rom_hal_timer_capture_init(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, stCapInit_t* pstCapInit);
```

Description

Initialize the indicated timer function of capture.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which channel will be configured, refer to the EN_TIMER_CH_T enumeration definition.
pstCapInit	Capture init struct, refer to the stCapInit_t enumeration definition.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hal_timer_enable_capture

Function Prototype

```
EN_ERR_STA_T rom_hal_timer_enable_capture(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh);
```

Description

Enable the timer capture function.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which channel will be enabled, refer to the EN_TIMER_CH_T enumeration definition.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hal_timer_disable_capture

Function Prototype

```
EN_ERR_STA_T rom_hal_timer_disable_capture(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh);
```

Description

Disable the timer capture function.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which channel will be disabled, refer to the EN_TIMER_CH_T enumeration definition.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

GPTM Examples

The main functions of the GPTM are described in detail in the first section, and this section develops the corresponding DEMO based on the previous two sections.

Counter Mode Example Code

The example takes GPTM2 as an example and carries out the implementation of the function of the cnt module in both overflow and compare modes, with level flipping when an interrupt is reached.

Interrupt Handler

```
static void gtim2_irq_handler(void)
{
    uint32_t u32IntFlag = 0;
    rom_hw_timer_get_interrupt_flag(TIMER2, TIMER_CHA, &u32IntFlag);
    rom_hw_timer_clear_interrupt_flag(TIMER2, TIMER_CHA, u32IntFlag);
    if(u32IntFlag & TIMER_INT_MATCH)
    {
        rom_hw_gpio_toggle_pin_output_level(GPIO_PORT_TIM2_CHA, GPIO_PIN_TIM2_CHA);
    }
    u32IntFlag = 0;
    rom_hw_timer_get_interrupt_flag(TIMER2, TIMER_CHB, &u32IntFlag);
    rom_hw_timer_clear_interrupt_flag(TIMER2, TIMER_CHB, u32IntFlag);
    if(u32IntFlag & TIMER_INT_OVERFLOW)
    {
        rom_hw_gpio_toggle_pin_output_level(GPIO_PORT_TIM2_CHB, GPIO_PIN_TIM2_CHB);
    }
}
```

GPTM Counter Mode Example Code

```
static void gtim2_mode_set(void)
{
    uint8_t u8Status = 0;
    stTimerInit_t pstTimerInit;
    pstTimerInit.u32Compare = 0;
    pstTimerInit.u8CounterMode = 0;
    pstTimerInit.u8Prescale = 0;
    stTimerInit_t pstTimerInit_A;
    pstTimerInit_A.u32Compare = 32767;
    pstTimerInit_A.u8CounterMode = 1;
    pstTimerInit_A.u8Prescale = 0;
    // Enable GPTM2 Clock
    rom_hal_timer_enable_clk(TIMER2);
    // Init GPTM2 CHB and CHA(Compare, Counter Mode and Prescale)
    rom_hal_timer_init(TIMER2, TIMER_CHB, &pstTimerInit);
    rom_hal_timer_init(TIMER2, TIMER_CHA, &pstTimerInit_A); 20.
    // Set GPTM2 CHB INT of Overflow and CHA INT of compare
    rom_hw_timer_enable_interrupt(TIMER2, TIMER_CHB, TIMER_INT_OVERFLOW);
    rom_hw_timer_enable_interrupt(TIMER2, TIMER_CHA, TIMER_INT_MATCH); 24.
    // Enable GPTM2 Peri INT
    rom_hw_sys_ctrl_enable_peri_int(SYS_CTRL_MP, TIMER2_IRQ); 27.
    // Register and Enable INT
    g_periIrqFuncTable[TIMER2_IRQ] = gtim2_irq_handler;
    NVIC_ClearPendingIRQ (TIMER2_IRQ);
    NVIC_SetPriority (TIMER2_IRQ, 0x3);
    NVIC_EnableIRQ (TIMER2_IRQ);
}
```



```
    __enable_irq();  
    // START GPTM2 CHB and CHA  
    rom_hw_timer_start(TIMER2, TIMER_CHB);  
    rom_hw_timer_start(TIMER2, TIMER_CHA);  
}
```

PWM Mode Example Code

This DEMO uses GPTM2 to generate PWM waves with different duty cycles on the CHA channel and CHB channel.

Interrupt Handler

```
static void gtim3_irq_handler(void)  
{  
    uint32_t u32IntFlag = 0;  
    rom_hw_timer_get_interrupt_flag(TIMER2, TIMER_CHA, &u32IntFlag);  
    rom_hw_timer_clear_interrupt_flag(TIMER2, TIMER_CHA, u32IntFlag);  
}
```

Set PWM Mode

```
// GPIO PWM Init  
rom_hw_gpio_set_pin_pid(GPIO_PORT_TIM3_CHB, GPIO_PIN_TIM3_CHB, PID_GTIM_PWM3_CHB);  
rom_hw_gpio_set_pin_pid(GPIO_PORT_TIM3_CHA, GPIO_PIN_TIM3_CHA, PID_GTIM_PWM3_CHA);  
  
// GPTM3 PWM Init  
rom_hal_timer_pwm_init(TIMER3, TIMER_CHB, 16, 3000);  
rom_hal_timer_pwm_init(TIMER3, TIMER_CHA, 16, 7000);  
rom_hal_timer_set_pwm_polarity(TIMER3, TIMER_CHB, PWM_POL_RISING);  
rom_hal_timer_set_pwm_polarity(TIMER3, TIMER_CHA, PWM_POL_RISING);  
// Enable PWM Mode  
rom_hal_timer_enable_pwm(TIMER3, TIMER_CHB);  
rom_hal_timer_enable_pwm(TIMER3, TIMER_CHA);  
rom_hw_sys_ctrl_enable_peri_int(SYS_CTRL_MP, TIMER3_IRQ);  
g_periIrqFuncTable[TIMER3_IRQ] = gtim3_irq_handler;  
NVIC_ClearPendingIRQ (TIMER3_IRQ);  
NVIC_SetPriority (TIMER3_IRQ, 0x3);  
NVIC_EnableIRQ (TIMER3_IRQ);  
__enable_irq();  
// START GPTM3  
rom_hw_timer_start(TIMER3, TIMER_CHB);  
rom_hw_timer_start(TIMER3, TIMER_CHA);
```

4 System Tick Timer (STIM)

Introduction

The device has two 32-bit System Tick Timers (STIM). Each 32-bit timer has a 4-channel compare register.

The STIM works in the low speed clock domain, it can keep working in the Sleep Mode. The STIM introduces a flexible clock scheme that delivers the required functionality and performance while also minimizing power consumption.

Features

- 2 way independent STIM: STIM0 & STIM1
- Supports a low-speed 32 kHz clock as a clock source
- Each STIM supports 32-bit counter, 12-bit prescale
- Each STIM supports 3 types of interrupt and wakeup: overflow/compare/tick
- Each STIM supports 4-channel compare interrupt and sleep wakeup
- Supports CPU interrupts and PMU wake interrupts independently
- Supports configuration success flag checking

Note: The STIM peripherals rely on a low-frequency clock source to provide frequency. Using a clock source with a stable frequency can achieve better clock accuracy.

Functional Description

The STIM peripherals are composed of STIM0 and STIM1, the Reference Design Schematic for each is shown below. The whole STIM has two counters, eight compare, two tick and overflow.

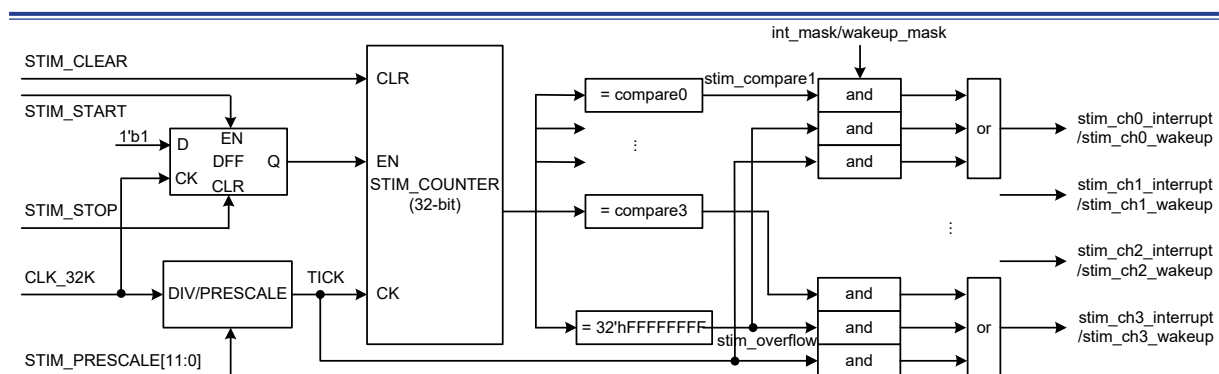


Figure 6. STIM Reference Design Schematic

Each STIM is a 32-bit counter that can only count up. It uses four clock sources, which are:

- RC 32K low frequency clock: RC_LCLK;
- DCXO 32K low frequency clock: DCXO_LCLK;

- RC high frequency clock frequency division: RC_HCLK_DIV_LCLK;
- DCXO high frequency clock division: DCXO_HCLK_DIV_LCLK.

Clock Accuracy and Prescale

The STIM of the device uses a clock source of 32768 Hz with a minimum time resolution of 30.517 μ s (a tick interval). The table below is the time interval for each tick at different frequency division coefficients. The overflow time is the time taken by the counter to count from 0 to 0xFFFF FFFF.

Table 4. Time Interval

PRESCALE	Tick Interval	Overflow
0	30.517 μ s	2183.275 minutes
$2^8 - 1$	7812.5 μ s	9320.676 hours
$2^{12} - 1$	125 ms	6213.784 days

Set PRESCALE as the frequency division coefficient, the time interval of each tick can be calculated as follows:

$$T_{RTC} = (\text{PRESCALE} + 1) / 32768, 0 \leq \text{PRESCALE} \leq 2^{12} - 1$$

Note: The PRESCALE register configuration of the device requires a trigger (Trig) to take effect.

Counter, Tick, Overflow and Compare

The counters of the two STIM peripherals of the device are both 32 bits long and can be counted independently in STIM0 and STIM1. Each STIM supports wakeup and interrupt configuration: 4-channel Compare, Overflow and Tick. When a STIM works, the counter increases according to the number of rising edges of the clock signal and triggers the Tick interrupt. The compare(n) interrupt is triggered when the counter register value reaches the compare(n) register value; When the counter register is full (0xFFFF FFFF), the overflow interrupt will be triggered and the counter value will be cleared, as shown below.

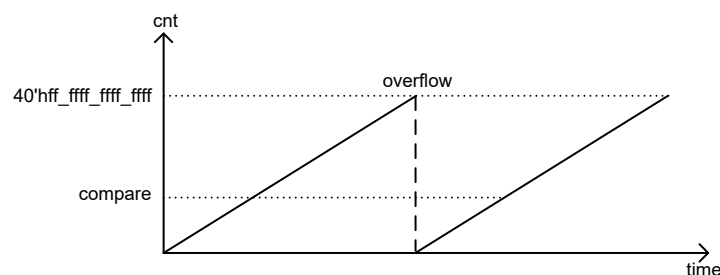


Figure 7. Counter Trigger Overflow

Note: The device disables the Tick, overflow, and Compare(n) interrupts by default and must be enabled when using them.

STIM APIs

STIM Interrupt APIs

rom_hw_stim_get_interrupt_flag

Function Prototype

```
EN_ERR_STA_T rom_hw_stim_get_interrupt_flag (stSTIM_Handle_t* pstSTIM, uint16_t* pu16Flag);
```

Description

Get the indicated timer interrupt flag (status) by reading the STIM_INT_FLAG register.

Parameter

Parameter	Description
pstSTIM	STIM handle, should be STIM0 / STIM1.
pu16Flag	Indicate which interrupt flag will be read.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_stim_clear_interrupt_flag

Function Prototype

```
EN_ERR_STA_T rom_hw_stim_clear_interrupt_flag (stSTIM_Handle_t* pstSTIM, uint16_t u16Flag);
```

Description

Clear the indicated STIM interrupt flag (status) by writing the STIM_INT_CLR register.

Parameter

Parameter	Description
pstSTIM	STIM handle, should be STIM0 / STIM1.
u16Flag	Indicate which flag will be cleared.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_stim_enable_interrupt

Function Prototype

```
EN_ERR_STA_T rom_hw_stim_enable_interrupt (stSTIM_Handle_t* pstSTIM, uint16_t u16IntEn);
```

Description

Enable the indicated STIM interrupt by writing the STIM_INT_EN register with indicated bits.

Parameter

Parameter	Description
pstSTIM	STIM handle, should be STIM0 / STIM1.
u16IntEn	Indicate which interrupt will be enabled, @ ref EN_STIM_INT_T.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_stim_disable_interrupt

Function Prototype

EN_ERR_STA_T rom_hw_stim_disable_interrupt (stSTIM_Handle_t* pstSTIM, uint16_t u16IntDis);

Description

Disable the indicated STIM interrupt by writing the STIM_INT_EN register with indicated bits.

Parameter

Parameter	Description
pstSTIM	STIM handle, should be STIM0 / STIM1.
u16IntDis	Indicate which interrupt will be disabled, @ ref EN_STIM_INT_T.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_stim_enable_tick_overflow_interrupt

Function Prototype

EN_ERR_STA_T rom_hw_stim_enable_tick_overflow_interrupt (stSTIM_Handle_t* pstSTIM, uint16_t u16IntEn);

Description

Enable the indicated STIM tick and overflow interrupt.

Parameter

Parameter	Description
pstSTIM	STIM handle, should be STIM0 / STIM1.
u16IntEn	Indicate which interrupt will be enabled, @ ref EN_STIM_INT_TICK_OVFLW_T.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_stim_disable_tick_overflow_interrupt

Function Prototype

EN_ERR_STA_T rom_hw_stim_disable_tick_overflow_interrupt (stSTIM_Handle_t* pstSTIM, uint16_t u16IntDis);

Description

Disable the indicated STIM tick and overflow interrupt.

Parameter

Parameter	Description
pstSTIM	STIM handle, should be STIM0 / STIM1.
u16IntDis	Indicate which interrupt will be disabled, @ ref EN_STIM_INT_TICK_OVFLW_T.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

STIM Wakeup & Status APIs

rom_hw_stim_enable_wakeup

Function Prototype

EN_ERR_STA_T rom_hw_stim_enable_wakeup (stSTIM_Handle_t* pstSTIM, uint16_t u16WakeupEn);

Description

Enable the indicated STIM wakeup interrupt.

Parameter

Parameter	Description
pstSTIM	STIM handle, should be STIM0 / STIM1.
u16WakeupEn	Indicate which wakeup interrupt will be enabled, @ ref EN_STIM_INT_WAKEUP_EN_T.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_stim_disable_wakeup

Function Prototype

EN_ERR_STA_T rom_hw_stim_disable_wakeup (stSTIM_Handle_t* pstSTIM, uint16_t u16WakeupDis);

Description

Disable the indicated STIM wakeup interrupt.

Parameter

Parameter	Description
pstSTIM	STIM handle, should be STIM0 / STIM1.
u16WakeupDis	Indicate which wakeup interrupt will be disabled, @ ref EN_STIM_INT_WAKEUP_EN_T.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_stim_start

Function Prototype

```
EN_ERR_STA_T rom_hw_stim_start (stSTIM_Handle_t* pstSTIM);
```

Description

Start the STIM counter.

Parameter

Parameter	Description
pstSTIM	STIM handle, should be STIM0 / STIM1.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_stim_stop

Function Prototype

```
EN_ERR_STA_T rom_hw_stim_stop (stSTIM_Handle_t* pstSTIM);
```

Description

Stop the STIM counter.

Parameter

Parameter	Description
pstSTIM	STIM handle, should be STIM0 / STIM1.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_stim_get_work_status

Function Prototype

```
EN_ERR_STA_T rom_hw_stim_get_work_status (stSTIM_Handle_t* pstSTIM, uint8_t* pu8Status);
```

Description

Get the STIM work status.

Parameter

Parameter	Description
pstSTIM	STIM handle, should be STIM0 / STIM1.
pu8Status	Point to save STIM work status.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

STIM Counter & Compare

rom_hw_stim_clear_count

Function Prototype

EN_ERR_STA_T rom_hw_stim_clear_count (stSTIM_Handle_t* pstSTIM);

Description

Clear the indicated STIM counter register.

Parameter

Parameter	Description
pstSTIM	STIM handle, should be STIM0 / STIM1.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_stim_set_count_overflow

Function Prototype

EN_ERR_STA_T rom_hw_stim_set_count_overflow (stSTIM_Handle_t* pstSTIM);

Description

The software generates an overflow event by setting the STIM_COUNTER_REG register to 0xFFFF FFF0. A STIM overflow interrupt will happen when the corresponding interrupt is enabled.

Parameter

Parameter	Description
pstSTIM	STIM handle, should be STIM0 / STIM1.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_stim_get_count

Function Prototype

```
EN_ERR_STA_T rom_hw_stim_get_count (stSTIM_Handle_t* pstSTIM, uint32_t* pu32Count);
```

Description

Get the indicated STIM counter register value.

Parameter

Parameter	Description
pstSTIM	STIM handle, should be STIM0 / STIM1.
pu32Count	Point to save STIM current counter value [31:0].

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_stim_set_prescale

Function Prototype

```
EN_ERR_STA_T rom_hw_stim_set_prescale (stSTIM_Handle_t* pstSTIM, uint16_t u16Prescale);
```

Description

Set the prescale value for the indicated STIM.

Parameter

Parameter	Description
pstSTIM	STIM handle, should be STIM0 / STIM1.
enCh	STIM compare channel, @ ref EN_STIM_CH_T.
u32Comp	STIM compare counter value [31:0].

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_stim_set_compare

Function Prototype

```
EN_ERR_STA_T rom_hw_stim_set_compare (stSTIM_Handle_t* pstSTIM, EN_STIM_CH_T enCh, uint32_t u32Comp);
```

Description

Set a 32-bit compare value for the indicated STIM. When the STIM counter is incremented to the same value with the compare value, an interrupt will happen when enabled.

Parameter

Parameter	Description
pstSTIM	STIM handle, should be STIM0 / STIM1.
enCh	STIM compare channel, @ ref EN_STIM_CH_T.
u32Comp	STIM compare counter value [31:0].

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_stim_get_compare

Function Prototype

EN_ERR_STA_T rom_hw_stim_get_compare (stSTIM_Handle_t* pstSTIM, EN_STIM_CH_T enCh, uint32_t* pu32Comp);

Description

Get the indicated STIM compare register STIM_COMP_REG value.

Parameter

Parameter	Description
pstSTIM	STIM handle, should be STIM0 / STIM1.
enCh	STIM compare channel, @ ref EN_STIM_CH_T.
pu32Comp	Point to save STIM compare counter value.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

STIM Example Code

The main functions of STIM peripherals have been described in the above section, and the following section will demonstrate the main functions of STIM.

STIM Wakeup & Interrupt Example Code

Example Code Project

This project demonstrates using STIM0 Channel0 to wake up the CPU and enter the interrupt handler.

```
uint8_t u8Status = 0;
/* 1. Enable stim0 clock gate first. */
hw_crg_enable_clk_gate (STIM0_CLK_GATE);
/* 2. Clear stim counter if needed. */
hw_stim_work_status (STIM0, &u8Status);
if (STIM_IS_WORKING == u8Status)
{
    hw_stim_stop (STIM0);
    hw_stim_clear_count (STIM0);
}
/* 3. Set stim prescale to 0(default). */
hw_stim_set_prescale (STIM0, 0);
/* 4. Configure STIM0 Channel0 Interrupt Handler*/
g_periIrqFuncTable[STIM0_IRQ0] = stim_stim0_ch0_handler;
NVIC_ClearPendingIRQ (STIM0_IRQ0);
NVIC_SetPriority (STIM0_IRQ0, 3);
NVIC_EnableIRQ (STIM0_IRQ0);
/* 5. Configure STIM0 interrupt and compare value. */
/* 5.1 Disable and clear interrupt flag*/
hw_stim_disable_wakeup (STIM0, STIM_INT_WAKEUP_EN_MASK);
```

```
hw_stim_disable_interrupt (STIM0, STIM_INT_MASK);
hw_stim_clear_interrupt_flag (STIM0, STIM_INT_TICK);
hw_stim_disable_tick_overflow_interrupt (STIM0, STIM_INT_TICK_OVFLW_MASK);
/* 5.2 Enable interrupt and wake up*/
hw_sys_ctrl_enable_peri_int (SYS_CTRL_MP, STIM0_IRQ0);
hw_stim_enable_wakeup (STIM0, STIM_CH0_INT_MATCH_WAKEUP);
hw_stim_enable_interrupt (STIM0, STIM_CH0_INT_MATCH);
/* 5.3 Set the compare0 channel0 to 200ms, the macro definition is used here to
    convert ms to a counter value */
hw_stim_set_compare(STIM0, STIM_CH0, STIM_MS_TO_CNT(LPWR_CLOCK_32K_HZ, 200));
/* 6. Configure stim0 channel0 wakeup source. */
hw_pmu_set_wakeup_source (0, LUT_TRIG_ID_OTHER, LUT_TRIG_ID_STIM0_CH0, LUT_
                        STIM0_CH0_ACT);
/* 7. Start stim0 counter. */
hw_stim_start(STIM0);
```

STIM Interrupt Handler Example Code

```
void stim_stim0_ch0_handler (void)
{
    uint32_t count = 0;
    uint16_t ul6Flag;
    /* Get stim0 channel0 interrupt flag and clear it. */
    hw_stim_get_interrupt_flag (STIM0, &ul6Flag);
    ul6Flag &= (STIM_CH0_INT_MATCH | STIM_INT_TICK | STIM_INT_OVERFLOW);
    hw_stim_clear_interrupt_flag(STIM0, ul6Flag);
    /* Get STIM0 Counter and print it. */
    hw_stim_get_count(STIM0, &count);
    PRINTF("STIM0 Counter is : %x", count);
}
```

STIM Overflow Interrupt Example Code

This project demonstrates how to use STIM0 overflow to wake up the CPU and enter the interrupt handler. The interrupt handler from the previous section is still used here.

Example Code Project

```
/* 1. Enable stim0 clock gate first. */
hw_crg_enable_clk_gate (STIM0_CLK_GATE);
/* 2. Configure Handler and Enable NVIC */
g_periIrqFuncTable[STIM0_IRQ0] = stim_stim0_ch0_handler;
NVIC_ClearPendingIRQ (STIM0_IRQ0);
NVIC_SetPriority (STIM0_IRQ0, 3);
NVIC_EnableIRQ (STIM0_IRQ0);
/* 3. Configure wake up source */
hw_pmu_set_wakeup_source (n, LUT_TRIG_ID_OTHER, LUT_TRIG_ID_STIM0_CH0, LUT_
                        STIM0_CH0_ACT);
/* 4. Enable stim0 channel 0 overflow interrupt and wakeup system. */
hw_stim_enable_wakeup (STIM0, STIM_INT_OVERFLOW_WAKEUP);
hw_stim_enable_interrupt (STIM0, STIM_INT_OVERFLOW);
hw_stim_enable_tick_overflow_interrupt (STIM0, STIM_CH0_INT_OVFLW);
/* 5. Trigger stim counter to 0xFFFFFFFF0 */
hw_stim_set_count_overflow(STIM0);
/* 6. Start STIM. */
hw_stim_start (STIM0);
```

5 Real Time Clock (RTC)

Introduction

The Real Time Clock (RTC) peripheral of the device provides a general purpose, low power timer on the Low-frequency Clock Source (LCLK). When the main power supply is normal, the main power supply supplies power to the RTC peripheral. When the main power supply is cut off, the RTC peripheral can be powered by an external lithium battery. During power switchover, the RTC data is always stored in the backup domain of the RTC.

Features

- Supports a low-speed 32 kHz clock as a clock source
- Supports 40-bit counter, 12-bit prescale
- Supports 3 types of interrupt: overflow/compare/tick
- Supports 4-channel interrupt and sleep wakeup:
 - Channel0: compare0/tick/overflow
 - Channel1: compare1/tick/overflow
 - Channel2: compare2/tick/overflow
 - Channel3: compare3/tick/overflow
- Supports start, stop, and clear Trig signal control
- Supports CPU interrupts and PMU wake interrupts independently
- Supports configuration success flag checking

Note: The RTC peripherals rely on a low-frequency clock source to provide frequency. Using a clock source with a stable frequency can achieve better clock accuracy.

Functional Description

The timer of the device RTC is a 40-bit counter that can only count up. It uses three clock sources, which are:

- 128 parts of a High Speed External clock: HSE / 128
- Low Speed Internal clock: LSI
- Low Speed External clock: LSE

The shutdown of the main power supply will affect the HSE frequency division and LSI clock sources, which cannot ensure the normal operation of the RTC when the power supply is shut down. Therefore, the RTC usually uses a low-speed external clock source LSE. The following diagram shows the device RTC Reference Design Schematic.

RTC Power Structure

[illegible]

May 12, 2025

Clock Accuracy and Prescale

The device RTC peripheral uses a clock source of 32768 Hz with a minimum time resolution of 30.517 μ s (a tick interval). Below is the time interval for each tick at different frequency division coefficients. The overflow time is the time taken by the counter to count from 0 to 0xFF FFFF FFFF.

PRESCALE	Tick Interval	Overflow
0	30.517 μ s	9315 hours
$2^8 - 1$	7812.5 μ s	99420 days
$2^{12} - 1$	125 ms	1590728 days

Set PRESCALER as the frequency division coefficient, the time interval of each tick can be calculated as follows:

$$T_{RTC} = (\text{PRESCALER} + 1) / 32768, 0 \leq \text{PRESCALER} \leq 2^{12} - 1$$

Note: The PRESCALE register configuration of the device requires a trigger (Trig) to take effect.

Counter, Tick, Overflow and Compare

The counter register length of the device RTC peripherals is 40-bit, and 4 channels of Compare, Overflow and tick interrupt or wakeup can be configured at the same time. The following diagram shows the Counter work process. When the RTC works, the Counter increases according to the number of rising edges of the clock signal and triggers the Tick interrupt. The compare(n) interrupt is triggered when the counter register value reaches the compare(n) register value; When the counter register is full (0xFF FFFF FFFF), the overflow interrupt will be triggered and the counter value will be cleared, as shown below.

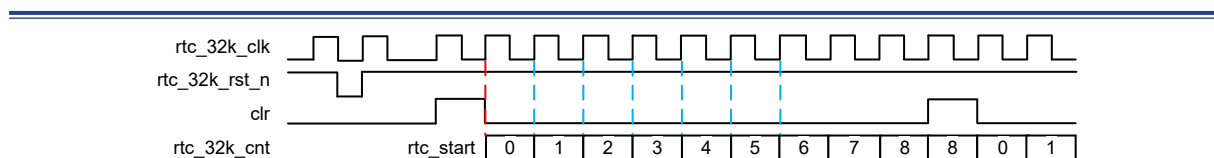


Figure 10. Counter Work Process

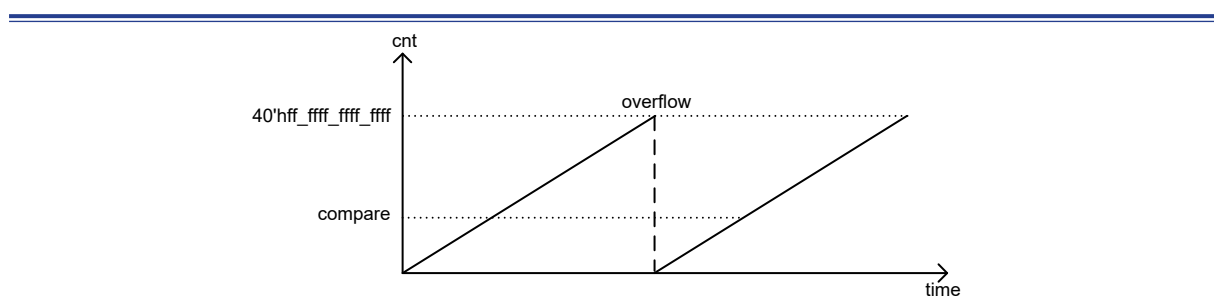


Figure 11. Counter Trigger Overflow

Note: The device disables the Tick, overflow, and Compare(n) interrupts by default and must be enabled when using them.

RTC APIs

RCT Configuration APIs

rom_hw_rtc_set_rtc_clk_src

Function Prototype

```
EN_ERR_STA_T rom_hw_rtc_set_rtc_clk_src (EN_RTC_CLK_SRC_T enSrc);
```

Description

Select the RTC_CLK clock source.

Parameter

Parameter	Description
enSrc	RTC clock source selection, @ ref EN_RTC_CLK_SRC_T.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_rtc_set_prescale

Function Prototype

```
EN_ERR_STA_T rom_hw_rtc_set_prescale (uint16_t u16Prescale);
```

Description

Set the RTC clock prescale value.

Parameter

Parameter	Description
u16Prescale	Prescale value.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_rtc_set_ldo_ret_output_voltage

Function Prototype

```
EN_ERR_STA_T rom_hw_rtc_set_ldo_ret_output_voltage (EN_RTC_LDO_RET_VOLT_T enVolt);
```

Description

Configure the VDD_RET output voltage. When the CPU gets into the sleep mode, the retention LDO will work and output the VDD_RET voltage to keep the system in the sleep mode.

Parameter

Parameter	Description
enVolt	Configure the LDO_RET (VDDRTC) output voltage, @ ref EN_RTC_LDO_RET_VOLT_T.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_rtc_pdw_reset

Function Prototype

EN_ERR_STA_T rom_hw_rtc_pdw_reset (void);

Description

Reset the RTC module power.

Parameter

None.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

RTC Interrupt APIs

rom_hw_rtc_get_interrupt_flag

Function Prototype

EN_ERR_STA_T rom_hw_rtc_get_interrupt_flag (uint16_t *pu16Msk);

Description

Get the indicated RTC interrupt flag (status) by reading the RTC_INT_FLAG register.

Parameter

Parameter	Description
pu16Flag	Indicate which interrupt flag will be read.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_rtc_clear_interrupt_flag

Function Prototype

EN_ERR_STA_T rom_hw_rtc_clear_interrupt_flag (uint16_t u16Msk)

Description

Clear the indicated RTC interrupt flag (status) by writing the RTC_INT_CLR register.

Parameter

Parameter	Description
u16Msk	Indicate which flag will be cleared.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_rtc_enable_interrupt

Function Prototype

```
EN_ERR_STA_T rom_hw_rtc_enable_interrupt (uint16_t u16IntEn);
```

Description

Enable the indicated RTC interrupt by writing RTC_INT_EN register with indicated bits.

Parameter

Parameter	Description
u16IntEn	Indicate which interrupt will be enabled.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_rtc_disable_interrupt

Function Prototype

```
EN_ERR_STA_T rom_hw_rtc_disable_interrupt (uint16_t u16IntDis);
```

Description

Disable the RTC interrupt by writing RTC_INT_EN register with indicated bits.

Parameter

Parameter	Description
u16IntDis	Indicate which interrupt will be disabled.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_rtc_enable_tick_overflow_interrupt

Function Prototype

```
EN_ERR_STA_T rom_hw_rtc_enable_tick_overflow_interrupt (uint16_t u16IntEn);
```

Description

Enable the indicated RTC tick and overflow interrupt.

Parameter

Parameter	Description
u16IntEn	Indicate which interrupt will be enabled, @ ref EN_RTC_INT_T.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_rtc_disable_tick_overflow_interrupt

Function Prototype

```
EN_ERR_STA_T rom_hw_rtc_disable_tick_overflow_interrupt (uint16_t u16IntDis);
```

Description

Disable the indicated RTC tick and overflow interrupt.

Parameter

Parameter	Description
u16IntDis	Indicate which interrupt will be disabled, @ ref EN_RTC_INT_T.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_rtc_enable_wakeup

Function Prototype

```
EN_ERR_STA_T rom_hw_rtc_enable_wakeup (uint16_t u16WakeupEn);
```

Description

Enable the indicated RTC wakeup interrupt. The CPU will be woken up from the sleep mode after the corresponding interrupt is enabled.

Parameter

Parameter	Description
u16WakeupEn	Indicate which wakeup interrupt will be enabled, @ ref EN_RTC_INT_WAKEUP_EN_T.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_rtc_disable_wakeup

Function Prototype

```
EN_ERR_STA_T rom_hw_rtc_disable_wakeup (uint16_t u16WakeupDis);
```

Description

Disable the indicated RTC wakeup interrupt.

Parameter

Parameter	Description
u16WakeupDis	Indicate which wakeup interrupt will be disabled, @ ref EN_RTC_INT_WAKEUP_EN_T.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

RTC Status APIs

rom_hw_rtc_start

Function Prototype

EN_ERR_STA_T rom_hw_rtc_start (void);

Description

Start the RTC counter.

Parameter

None.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_rtc_stop

Function Prototype

EN_ERR_STA_T rom_hw_rtc_stop (void);

Description

Stop the RTC counter.

Parameter

None.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_rtc_get_work_status

Function Prototype

EN_ERR_STA_T rom_hw_rtc_get_work_status (uint8_t* pu8Status);

Description

Get the RTC work status.

Parameter

Parameter	Description
pu8Status	Point to save RTC work status.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_rtc_get_wakeup_status

Function Prototype

EN_ERR_STA_T rom_hw_rtc_get_wakeup_status (uint8_t* pu8Status);

Description

Get the RTC wakeup source work status, the PMU enters low power consumption only when it is in the stop state.

Parameter

Parameter	Description
pu8Status	Point to save RTC wakeup source work status, @ ref EN_RTC_WAKEUP_STATUS_T.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_rtc_set_sw_flag

Function Prototype

EN_ERR_STA_T rom_hw_rtc_set_sw_flag (uint32_t u32SwFlag);

Description

Configure software flag. Check if the RTC module is initialized or not. When the VDDR is powered down, this flag will be retained.

Parameter

Parameter	Description
pu8Status	Software flag.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

RTC Counter & Compare

rom_hw_rtc_get_count

Function Prototype

EN_ERR_STA_T rom_hw_rtc_get_count (uint8_t* pu8Hi, uint32_t* pu32Lo);

Description

Get the RTC current counter value.

Parameter

Parameter	Description
pu8Hi	Point to save RTC current counter value [39:32].
pu32Lo	Point to save RTC current counter value [31:0].

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_rtc_get_count64

Function Prototype

```
EN_ERR_STA_T rom_hw_rtc_get_count64 (uint64_t* pu64Count);
```

Description

Get the RTC current counter value.

Parameter

Parameter	Description
pu64Count	Point to save RTC current counter value [39:0].

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_rtc_clear_count

Function Prototype

```
EN_ERR_STA_T rom_hw_rtc_clear_count (void);
```

Description

Clear the RTC counter.

Parameter

None.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_rtc_set_count_overflow

Function Prototype

```
EN_ERR_STA_T rom_hw_rtc_set_count_overflow (void);
```

Description

Set the RTC counter to 0xFFFFFFFF0 to trigger an overflow interrupt quickly.

Parameter

None.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_rtc_set_compare

Function Prototype

```
EN_ERR_STA_T rom_hw_rtc_set_compare (EN_RTC_CH_T enCh, uint8_t u8Hi, uint32_t u32Lo);
```

Description

Set the RTC compare counter value. When the RTC is incremented to the same value with the compare counter value, the RTC will generate an interrupt when enabled.

Parameter

Parameter	Description
enCh	RTC compare channel, @ ref EN_RTC_CH_T.
u8Hi	RTC compare counter value [39:32].
u32Lo	RTC compare counter value [31:0].

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_rtc_set_compare64

Function Prototype

```
EN_ERR_STA_T rom_hw_rtc_set_compare64 (EN_RTC_CH_T enCh, uint64_t u64Compare);
```

Description

Set the RTC compare counter value. When the RTC is incremented to the same value with the compare counter value, the RTC will generate an interrupt when enabled.

Parameter

Parameter	Description
enCh	RTC compare channel, @ ref EN_RTC_CH_T.
u64Compare	RTC compare counter value [39:0].

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_rtc_get_compare

Function Prototype

```
EN_ERR_STA_T rom_hw_rtc_get_compare (EN_RTC_CH_T enCh, uint8_t* pu8Hi, uint32_t* pu32Lo);
```

Description

Get the RTC compare counter value.

Parameter

Parameter	Description
enCh	RTC compare channel, @ ref EN_RTC_CH_T.
pu8Hi	Point to save RTC compare counter value [39:32].
pu32Lo	Point to save RTC compare counter value [31:0].

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_rtc_get_compare64

Function Prototype

EN_ERR_STA_T rom_hw_rtc_get_compare64 (EN_RTC_CH_T enCh, uint64_t* pu64Compare);

Description

Get the RTC compare counter value.

Parameter

Parameter	Description
enCh	RTC compare channel, @ ref EN_RTC_CH_T.
pu64Compare	Point to save RTC compare counter value [39:0].

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

RTC Example Code

The main functions of RTC peripherals have been described in the above section, and the following section will demonstrate the main functions of RTC.

RTC Start Work Example Code

When using RTC peripherals, it is necessary to set the VDDRTC voltage and start the clock. This project demonstrates the clock settings when the RTC works normally.

```
/* Open the system APB clock & RC clock */
rom_hw_crg_enable_clk_gate (CRG_RTC_APB_CLK_GATE);
rom_hw_crg_enable_clk_gate (CRG_RTC_RC_LCLK_GATE);
/* Set the voltage of the VDDRTC */
patch_hw_rtc_set_ldo_ret_output_voltage(EN_RTC_LDO_RET_950mV);
/* Set RC 32K as the RTC clock source */
patch_hw_rtc_set_rtc_clk_src(EN_RTC_CLK_SRC_RC_LCLK);
/* RTC start to work */
patch_hw_rtc_start();
```

RTC Interrupt Example Code

This sample project is designed to test whether the RTC can respond to the Compare interrupt by taking the current Counter value and setting Compare1 to the current Counter + 1000. The examples are as follows.

RTC Interrupt Example Code

```
void RTC_Interrupt (void)
{
    uint64_t now_counter = 0;
    uint64_t compare_val = 1000;
    /* Stop RTC counter */
    patch_hw_rtc_stop();
    /* Set prescale value */
```

```
patch_hw_rtc_set_prescale(0);
/* Enable RTC channel 1 interrupt */
patch_hw_rtc_enable_interrupt(RTC_CH1_INT_MATCH);
/* Get counter value */
patch_hw_rtc_get_count64(&now_counter);
/* Set compare1 value, Interrupt will be triggered after 1000 counters */
patch_hw_rtc_set_compare64(RTC_CH1, now_counter + compare_val);
/* Configure for RTC interrupt handler */
g_periIrqFuncTable[RTC_CH1_IRQ] = RTC_Interrupt_Handler;
/* Enable system interrupt */
NVIC_ClearPendingIRQ (RTC_CH1_IRQ);
NVIC_SetPriority (RTC_CH1_IRQ, 3);
NVIC_EnableIRQ (RTC_CH1_IRQ);
__enable_irq();
/* RTC start to work */
patch_hw_rtc_start();
}
```

RTC Interrupt Handler Example Code

```
void RTC_Interrupt_Handler(void)
{
    uint16_t int_flag = 0;
    uint64_t compare_val = 0;
    uint64_t counter_val = 0;
    /* Clear RTC Interrupt flag */
    patch_hw_rtc_get_interrupt_flag(&int_flag);
    patch_hw_rtc_clear_interrupt_flag(int_flag);
    /* Gets the value of the current counter */
    patch_hw_rtc_get_count64(&counter_val);
    PRINTF("The counter value is: %llx \n", counter_val);
}
```

Wakeup Example Code

The following example code shows how the RTC wakes up the CPU in the SLEEP mode.

```
void RTC_Wake(void)
{
    uint64_t now_counter = 0;
    uint64_t compare_val = 1000;
    patch_hw_rtc_stop();
    patch_hw_rtc_set_prescale(0);
    patch_hw_rtc_enable_interrupt(RTC_CH1_INT_MATCH);
    patch_hw_rtc_get_count64(&now_counter);
    patch_hw_rtc_set_compare64(RTC_CH1, now_counter + compare_val);
    /* Set RTC CH1 as the wakeup source */
    rom_hw_pmu_set_wakeup_source (9, LUT_TRIG_ID_OTHER, LUT_TRIG_ID_RTC_CH1,
                                   LUT_ACT_PD_SYS_ON | LUT_ACT_MP_IRQ_EN);

    /* Enable wakeup */
    patch_hw_rtc_enable_wakeup(RTC_CH1_INT_MATCH_WAKEUP);
    patch_hw_rtc_start();
}
```


6 Universal Asynchronous Receiver Transmitter (UART)

Introduction

The UART is compliant to the industry-standard 16550 and is used for serial communication with a peripheral data set. Data is written from a master (CPU/DMA) over the APB bus to the UART and it is converted to serial form and transmitted to the destination device. Serial data is also received by the UART and stored for the master (CPU/DMA) to read back. These UARTs support hardware flow control signals (RTS, CTS).

Features

- 8-byte transmit and receive FIFOs
- Hardware flow control support (CTS/RTS)
- Functionality based on the 16550 industry standard
- Programmable character properties, such as number of data bits per character (5~8), optional
- Parity bit (with odd/even/stick/no select) and number of stop bits (1, 1.5 or 2)
- Line break generation and detection
- RX timeout interrupt support
- Programmable serial data baud rate

Functional Description

The device UART functions mainly contain: UART initialization, UART configuration, serial data receiving/sending, interrupt configuration, etc.

UART APIs

UART Configure APIs

rom_hw_uart_deinit

Function Prototype

EN_ERR_STA_T rom_hw_uart_deinit(stUART_Handle_t* pstUART);

Description

Initializes the UART peripheral.

Parameter

Parameter	Description
pstUART	UART handle, should be UART0/UART1/UART2.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_uart_init

Function Prototype

```
EN_ERR_STA_T rom_hw_uart_init(stUART_Handle_t* pstUART, stUartInit_t* pstInitType);
```

Description

Initializes the UART mode according to the specified parameters in the stUartInit_t and create the associated handle.

Parameter

Parameter	Description
pstUART	UART handle, should be UART0 / UART1 / UART2.
pstInitType	Pointer to a stUartInit_t structure.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_uart_set_baudrate

Function Prototype

```
EN_ERR_STA_T rom_hw_uart_set_baudrate(stUART_Handle_t* pstUART, uint32_t u32BaudRate);
```

Description

Set the UART baud rate.

Parameter

Parameter	Description
pstUART	UART handle, should be UART0 / UART1 / UART2.
u32BaudRate	Baud rate.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_uart_set_data_sizes

Function Prototype

```
EN_ERR_STA_T rom_hw_uart_set_data_sizes(stUART_Handle_t* pstUART, EN_UART_DATA_SIZE_T enSize);
```

Description

Set the UART data bits. The data bits are upward compatible, the high data bits can correctly receive data with the low data bits. The maximum value of databit5 is 31, the maximum value of databit6 is 63, the maximum value of databit7 is 127 and the maximum value of databit8 is 255.

Parameter

Parameter	Description
pstUART	UART handle, should be UART0 / UART1 / UART2.
enSize	UART data size , @ ref EN_UART_DATA_SIZE_T.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_uart_set_stop_sizes

Function Prototype

EN_ERR_STA_T rom_hw_uart_set_stop_sizes(stUART_Handle_t* pstUART, EN_UART_STOP_SIZE_T enStopbits);

Description

Set the UART stop bit.

Parameter

Parameter	Description
pstUART	UART handle, should be UART0 / UART1 / UART2.
enStopbits	UART stop bit, @ ref EN_UART_STOP_SIZE_T.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_uart_set_parity

Function Prototype

EN_ERR_STA_T rom_hw_uart_set_parity(stUART_Handle_t* pstUART, EN_UART_PARITY_BITS_T enParity);

Description

Set the UART parity.

Parameter

Parameter	Description
pstUART	UART handle, should be UART0 / UART1 / UART2.
enParity	UART parity, @ ref EN_UART_PARITY_BITS_T.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_uart_set_mode

Function Prototype

EN_ERR_STA_T rom_hw_uart_set_mode(stUART_Handle_t* pstUART, EN_UART_ENDIAN_T enMode);

Description

Set the UART endian mode.

Parameter

Parameter	Description
pstUART	UART handle, should be UART0 / UART1 / UART2.
enMode	UART endian mode, @ ref EN_UART_ENDIAN_T.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_uart_line_break_enable

Function Prototype

```
EN_ERR_STA_T rom_hw_uart_line_break_enable(stUART_Handle_t* pstUART, EN_UART_LB_EN_T enEn);
```

Description

Configure the UART send line break function.

Parameter

Parameter	Description
pstUART	UART handle, should be UART0 / UART1 / UART2.
enEn	Enable or disable TX line break function.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_uart_set_rx_timeout

Function Prototype

```
EN_ERR_STA_T rom_hw_uart_set_rx_timeout(stUART_Handle_t* pstUART, uint8_t u8SymbolNum);
```

Description

Set the UART RX timeout.

Parameter

Parameter	Description
pstUART	UART handle, should be UART0 / UART1 / UART2.
u8SymbolNum	Timeout value, unit: ms

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_uart_flow_enable

Function Prototype

```
EN_ERR_STA_T rom_hw_uart_flow_enable(stUART_Handle_t* pstUART, EN_UART_FLOW_EN_T enEn);
```

Description

Configure the UART flow function.

Parameter

Parameter	Description
pstUART	UART handle, should be UART0 / UART1 / UART2.
enEn	Enable or disable flow function, @ ref EN_UART_FLOW_EN_T.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_uart_set_rts_thld

Function Prototype

```
EN_ERR_STA_T rom_hw_uart_set_rts_thld(stUART_Handle_t* pstUART, uint8_t u8Thld);
```

Description

Set the RTS threshold.

Parameter

Parameter	Description
pstUART	UART handle, should be UART0 / UART1 / UART2.
u8Thld	Threshold for UART RTS.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

UART Interrupt APIs

rom_hw_uart_get_interrupt_flag

Function Prototype

```
EN_ERR_STA_T rom_hw_uart_get_interrupt_flag(stUART_Handle_t* pstUART, uint16_t* pu16Falg);
```

Description

Get the UART interrupt flag (status).

Parameter

Parameter	Description
pstUART	UART handle, should be UART0 / UART1 / UART2.
pu16Falg	Indicate which interrupt flag will be read.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_uart_enable_interrupt

Function Prototype

```
EN_ERR_STA_T rom_hw_uart_enable_interrupt(stUART_Handle_t* pstUART, uint16_t u16Msk);
```

Description

Set the UART interrupt enable.

Parameter

Parameter	Description
pstUART	UART handle, should be UART0 / UART1 / UART2.
u16Msk	Indicate which interrupt will be enabled. Bit = 1 means enable Bit = 0 means no impact

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_uart_disable_interrupt

Function Prototype

```
EN_ERR_STA_T rom_hw_uart_disable_interrupt(stUART_Handle_t* pstUART, uint16_t u16Msk);
```

Description

Set the UART interrupt disable.

Parameter

Parameter	Description
pstUART	UART handle, should be UART0 / UART1 / UART2.
u16Msk	Indicate which interrupt will be disabled. Bit = 1 means disable Bit = 0 means no impact

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_uart_clear_interrupt_flag

Function Prototype

```
EN_ERR_STA_T rom_hw_uart_clear_interrupt_flag(stUART_Handle_t* pstUART, uint16_t u16Msk);
```

Description

Clear the indicated UART interrupt flag.

Parameter

Parameter	Description
pstUART	UART handle, should be UART0 / UART1 / UART2.
u16Msk	Indicate which flag will be cleared.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

UART FIFO APIs

rom_hw_uart_set_txfifo_thld

Function Prototype

```
EN_ERR_STA_T rom_hw_uart_set_txfifo_thld(stUART_Handle_t* pstUART, uint8_t u8Thld);
```

Description

Set the TX FIFO under threshold level. When the TX FIFO is under this threshold, it will trigger an interrupt.

Parameter

Parameter	Description
pstUART	UART handle, should be UART0 / UART1 / UART2.
u8Thld	Threshold for UART TX FIFO.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_uart_set_rxfifo_thld

Function Prototype

```
EN_ERR_STA_T rom_hw_uart_set_rxfifo_thld(stUART_Handle_t* pstUART, uint8_t u8Thld);
```

Description

Set the RX FIFO over threshold level. When the RX FIFO is over the threshold, it will trigger an interrupt.

Parameter

Parameter	Description
pstUART	UART handle, should be UART0 / UART1 / UART2.
u8Thld	Threshold for UART RX FIFO.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_uart_clear_rxfifo

Function Prototype

```
EN_ERR_STA_T rom_hw_uart_clear_rxfifo(stUART_Handle_t* pstUART);
```

Description

Configure the UART clear FIFO function.

Parameter

Parameter	Description
pstUART	UART handle, should be UART0 / UART1 / UART2.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_uart_get_txfifo_cnt

Function Prototype

```
EN_ERR_STA_T rom_hw_uart_get_txfifo_cnt(stUART_Handle_t* pstUART, uint8_t* pu8Cnt);
```

Description

Get the UART TX FIFO count.

Parameter

Parameter	Description
pstUART	UART handle, should be UART0 / UART1 / UART2.
pu8Cnt	TX FIFO count values.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_uart_get_rxfifo_cnt

Function Prototype

```
EN_ERR_STA_T rom_hw_uart_get_rxfifo_cnt(stUART_Handle_t* pstUART, uint8_t* pu8Cnt);
```

Description

Get the UART RX FIFO count.

Parameter

Parameter	Description
pstUART	UART handle, should be UART0 / UART1 / UART2.
pu8Cnt	RX FIFO count values.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

UART RX/TX APIs

rom_hw_uart_get_byte

Function Prototype

```
EN_ERR_STA_T rom_hw_uart_get_byte(stUART_Handle_t* pstUART, uint8_t* pu8Data);
```

Description

Get a byte of data in the non-blocking mode.

Parameter

Parameter	Description
pstUART	UART handle, should be UART0 / UART1 / UART2.
pu8Data	Pointer to data buffer.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_uart_receive

Function Prototype

```
EN_ERR_STA_T rom_hw_uart_receive(stUART_Handle_t* pstUART, uint8_t* pu8Buf, uint8_t u8BufSize);
```

Description

Receive an amount of data in the non-blocking mode.

Parameter

Parameter	Description
pstUART	UART handle, should be UART0 / UART1 / UART2.
pu8Data	Pointer to data buffer.
u16Len	Amount of data to be received.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_uart_send_byte

Function Prototype

```
EN_ERR_STA_T rom_hw_uart_send_byte(stUART_Handle_t* pstUART, uint8_t u8Data);
```

Description

Transmit single data through the UART when the TX FIFO is not full, blocking while sending.

Parameter

Parameter	Description
pstUART	UART handle, should be UART0 / UART1 / UART2.
u8Data	The data to be transmitted.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

rom_hw_uart_transmit

Function Prototype

```
EN_ERR_STA_T rom_hw_uart_transmit(stUART_Handle_t* pstUART, uint8_t* pu8Buf, uint16_t u16Len);
```

Description

Transmit an amount of data in the blocking mode.

Parameter

Parameter	Description
pstUART	UART handle, should be UART0 / UART1 / UART2.
pu8Data	Pointer to data buffer.
u16Len	Amount of data to be sent.

Return

HW status	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
-----------	--

UART Example Code

Example of Application Using UART1

Initialize the Serial Port Pin

To initialization the serial port pin, step 1: define the GPIO for the serial port. Step 2: initialize the clock of the serial port. Step 3: Set the serial port GPIO. The code is as follows:

```
#define GPIO_PIN_UART1_TX(GPIO_PIN_10)
#define GPIO_PIN_UART1_RX(GPIO_PIN_11)
static void uart_config(void)
{
    // Initialize the uart clock
    rom_hw_crg_enable_clk_gate(CRG_UART1_CLK_GATE);
    // Initialize the serial port pin
    patch_hw_gpio_set_pin_pid (GPIOA, GPIO_PIN_UART1_TX, PID_UART1_TXD);
    patch_hw_gpio_set_pin_pid (GPIOB, GPIO_PIN_UART1_RX, PID_UART1_RXD);
    // Initializing the serial port
    uart_init(UART1);
}
```

Initialize UART

To initialize the UART, step 1: Initialize the UART handle for register the interrupt function. Step 2: Initialize the serial port structure, set the baud rate, FIFO, data bits and parity. Step 3: Set the interrupt receiving mode. Step 4: Register the interrupt service function. Step 5: Enable timeout interrupts. The code is as follows:

```
static void uart_init(pstUART_Handle_t UART_Handle)
{
    // Initialize the uart handle
    pstUART_Handle_t g_pu32UartHandle = UART_Handle;
    // Initialize the initial uart
    struct stUartInit_t stUartInit;
    // Set the communication baud rate
    stUartInit.u32UartBaudRate = UART_BAUDRATE_921600;
    // Set the default configuration of the serial port
    // TxFifoThld and RxFifoThld is 8, StopBit is 1, 8bit, no parity, little-
    ending
    stUartInit.unUartCfg.u32UartCfg = UART_INIT_DEFAULT(UART_PARITY_NONE);
    // Write the default configuration of the serial port
    rom_hw_uart_init(g_pu32UartHandle, &stUartInit);
    // Receive data with a timeout interrupt
    rom_hw_uart_set_rx_timeout(g_pu32UartHandle, 0xFF);
    // Register the interrupt service function
    uart_nvic_init(UART_Handle);
    // Enable timeout interrupts
    rom_hw_uart_enable_interrupt(g_pu32UartHandle, UART_INT_RX_TIMEOUT);
}
```

Register the Interrupt Service Function

When there is UART data received, it will enter the UART interrupt callback function. The interrupt service function is as follows:

```
static void uart_nvic_init(pstUART_Handle_t UART_Handle)
{
    // Clear the interrupt status flag
    rom_hw_uart_clear_interrupt_flag(UART_Handle, 0x1fff);
    // Register the interrupt callback function.
    g_periIrqFuncTable[UART1_IRQ] = (PERI_IRQ_FUNC)UART1_IRQ_Handler;
    // Clear the interrupt
    NVIC_ClearPendingIRQ (UART1_IRQ);
    // Set interrupt priority
    NVIC_SetPriority (UART1_IRQ, 0x3);
    // enable interrupt
    NVIC_EnableIRQ (UART1_IRQ);
    // Here is divided into cortex_M33 and cortex_M0+
    #ifdef MAIN_PROCESSOR
    rom_hw_sys_ctrl_enable_peri_int(SYS_CTRL_MP, UART1_IRQ);
    #else
    rom_hw_sys_ctrl_enable_peri_int(SYS_CTRL_CP, UART1_IRQ);
    #endif
    __enable_irq();
}
```

Receive Data

Data is usually received in the UART interrupt callback function. When the data is judged to be available, it can be received. The receive data code is as follows:

```
static void UART1_IRQ_Handler(void)
{
    static uint8_t u8Cnt = 0;
    // Determines if data is available.
    while(UART1->UART_RXFIFO_CNT & 0x1F)
    {
        // Save data
        g_PCMsg.Data[u8Cnt++] = UART1 -> UART_RX_FIFO & 0xFF;
    }
    // Clear interruption flags.
    rom_hw_uart_clear_interrupt_flag(UART1, UART_INT_RX_TIMEOUT);
}
```

Send Data

This function can be used for single-byte sending. Refer to the previous section for detailed parameters.

```
rom_hw_uart_send_byte (stUART_Handle_t* pstUART, uint8_t u8Data);
```

7 IR Module

Introduction

The device can support IR encoding and IR decoding, both of which are completed by the GTIM part of the chip.

IR encode: The chip implements IR encoding through GTIM, and then the generated IR signal is sent through the connected IR diode.

IR decode: The chip contains an IR receiving circuit, which can receive IR signals, and then realize IR decoding through software.

Features

- 1-channel internal infrared transmission circuit
- 4-channel IR encoding
 - IR encoded signal can be output through GPIO
 - 4-channel IR emission can be realized by connecting an IR transmission circuit on the GPIO
- 4-channel IR trigger decode function
 - Support cnta/cntb to decode (support cnta and cntb to decode at the same time)
 - Support combined 32-bit for decoding

Functional Description

IR Encode

It is used to generate IR waveforms.

The IR waveform is formed by the combination of two PWM waveforms that generated by cnta and cntb of GTIM.

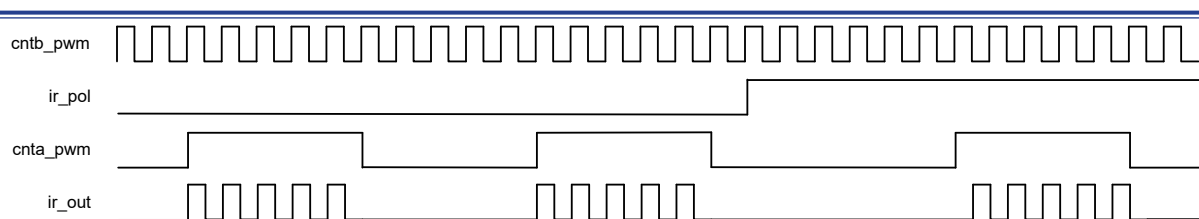


Figure 12. IR Encode

Use cntb to output a PWM with a specified frequency and a duty cycle of 50% as a carrier, and cnta to output the specified PWM as a data waveform signal, and then use the logical “and” to synthesize the required infrared signal from two PWM signals.

ir_pol can be used to invert the carrier signal cntb waveform to meet specific needs.

Note: There are four IR encoding channels in the device, but there is only one internal IR sending pin.

The IR encoded signal of the device can be output through GPIO, so the device can achieve the effect of four-channel IR emission by means of external hardware devices.

IR Decode

It is used for IR decoding to restore the IR signal with carrier to the level signal.

Each GTIM contains two channels (channel A and channel B), and there is a decode counter in each channel. When the waveform change time is within the configuration count time (decode interval), it outputs a high level. When the waveform change time exceeds the configuration time, it outputs a low level. (decode_gpout_pol can be configured to invert the output)

Two channels can also be combined into a 32-bit channel for use.

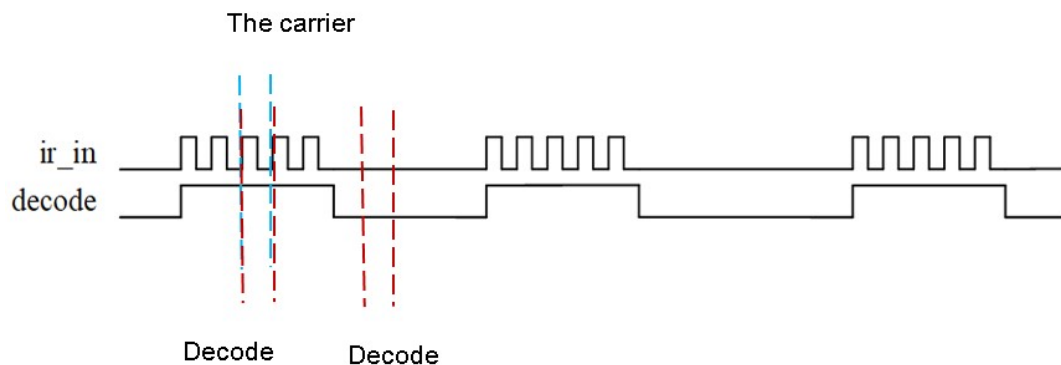


Figure 13. IR Decode

Note: The configuration count time is called decode interval in code configuration.

When it is detected that the input signal is high, the decode signal will output a high level, and then decode starts to count. If the waveform change of the input IR signal is detected in an interval, the decode will continue to output a high level. If no waveform change is detected within an interval, the decode will be pulled low immediately.

We need to ensure that the time of one change cycle of the input waveform is within the decode interval, and the decode interval is also as close as possible to the time of one change cycle of the input waveform to reduce the decoding error.

IR Module APIs

IR Encode HW APIs

rom_hw_ir_set_send_path

Function Prototype

```
EN_ERR_STA_T rom_hw_ir_set_send_path (stTIMER_Handle_t* pstIR, EN_IR_SEND_PATH_T enPath);
```

Description

Set the IR send signal path.

The EN_IR_SEND_PATH_T definition is as follows:

```
typedef enum
{
    IR_SEND_DEFAULT_PATH = 0,
    IR_SEND_BY_IR_TX_PIN = 1,
    IR_SEND_BY_GPIO = 2,
    IR_SEND_BY_BOTH = 3,
} EN_IR_SEND_PATH_T;
```

Parameter

Parameter	Description
pstIR	IR handle, should be IR0 / IR1 / IR2 / IR3.
enPath	IR send signal path, refer to the EN_IR_SEND_PATH_T definition.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_ir_set_pwm_current_compare_and_polarity

Function Prototype

```
EN_ERR_STA_T rom_hw_ir_set_pwm_current_compare_and_polarity (stTIMER_Handle_t* pstIR, uint16_t u16Compare, EN_PWM_POL_T enPol)
```

Description

Configure the IR TX parameters including current, PWM polarity etc.

Parameter

Parameter	Description
pstIR	IR handle, should be IR0 / IR1 / IR2 / IR3.
u16Compare	Compare value. The range is 0 ~ 65535.
enPol	PWM polarity, refer to the EN_PWM_POL_T definition.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_ir_pwm_set_next_compare

Function Prototype

EN_ERR_STA_T rom_hw_ir_pwm_set_next_compare (stTIMER_Handle_t* pstIR, uint16_t u16Compare, uint16_t u16LoCnt, uint16_t u16HiCnt);

Description

Configure the IR TX carrier PWM period and width.

Parameter

Parameter	Description
pstIR	IR handle, should be IR0 / IR1 / IR2 / IR3.
u16Compare	Compare value. The range is 0 ~ 65535.
u16HiCnt	Compare high counter. When the counter counts to the high counter value, the level will flip once. The range is 0 ~ 65535.
u16LoCnt	Compare low counter. When the counter counts to the low counter value, the level will flip once. The range is 0 ~ 65535.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_ir_enable

Function Prototype

EN_ERR_STA_T rom_hw_ir_enable (stTIMER_Handle_t* pstIR);

Description

Enable the GPTM working in IR function.

Parameter

Parameter	Description
pstIR	IR handle, should be IR0 / IR1 / IR2 / IR3.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_ir_disable

Function Prototype

EN_ERR_STA_T rom_hw_ir_disable (stTIMER_Handle_t* pstIR);

Description

Disable the GPTM IR function.

Parameter

Parameter	Description
pstIR	IR handle, should be IR0 / IR1 / IR2 / IR3.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_ir_set_polarity

Function Prototype

EN_ERR_STA_T rom_hw_ir_set_polarity (stTIMER_Handle_t* pstIR, EN_IR_POL_T enPol);

Description

Configure the IR TX output carrier polarity.

Parameter

Parameter	Description
pstIR	IR handle, should be IR0 / IR1 / IR2 / IR3.
enPol	IR polarity, refer to the EN_PWM_POL_T definition.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_ir_enable_channel

Function Prototype

EN_ERR_STA_T rom_hw_ir_enable_channel (stTIMER_Handle_t* pstIR, EN_IR_CH_T enCh);

Description

Enable IR channel.

Parameter

Parameter	Description
pstIR	IR handle, should be IR0 / IR1 / IR2 / IR3.
enCh	IR channel, refer to the EN_IR_CH_T enumeration definition.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_ir_disable_channel

Function Prototype

```
EN_ERR_STA_T rom_hw_ir_disable_channel (stTIMER_Handle_t* pstIR, EN_IR_CH_T enCh);
```

Description

Disable IR channel.

Parameter

Parameter	Description
pstIR	IR handle, should be IR0 / IR1 / IR2 / IR3.
enCh	IR channel, refer to the EN_IR_CH_T enumeration definition.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

IR Encode HAL APIs

rom_hal_ir_send_init

Function Prototype

```
EN_ERR_STA_T rom_hal_ir_send_init (stTIMER_Handle_t* pstIR, uint32_t u32Freq, uint16_t u16Duty, EN_IR_SEND_PATH_T enPath);
```

Description

Initialize the indicated IR timer clock.

Parameter

Parameter	Description
pstIR	IR handle, should be IR0 / IR1 / IR2 / IR3.
u32Frequency	PWM output frequency of carrier, unit: Hz
u8Duty	PWM duty of carrier

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hal_ir_start_send

Function Prototype

```
EN_ERR_STA_T rom_hal_ir_start_send (stTIMER_Handle_t* pstIR);
```

Description

Start IR sending.

Parameter

Parameter	Description
pstIR	IR handle, should be IR0 / IR1 / IR2 / IR3.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hal_ir_stop_send

Function Prototype

EN_ERR_STA_T rom_hal_ir_stop_send (stTIMER_Handle_t* pstIR);

Description

Stop IR sending.

Parameter

Parameter	Description
pstIR	IR handle, should be IR0 / IR1 / IR2 / IR3.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hal_ir_config_clock

Function Prototype

EN_ERR_STA_T rom_hal_ir_config_clock (stTIMER_Handle_t* pstIR, EN_IR_CH_T enCh, uint8_t u8Prescale);

Description

Configure an indicated channel prescale.

Parameter

Parameter	Description
pstIR	IR handle, should be IR0 / IR1 / IR2 / IR3.
enCh	IR channel.
u8Prescale	The divider for the input clock, the range is 0 ~ 15.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hal_ir_send_next_signal_level

Function Prototype

EN_ERR_STA_T rom_hal_ir_send_next_signal_level (stTIMER_Handle_t* pstIR, unIR_SendSignalData_t* punIrData);

Description

Configure the IR next period compare value.

Parameter

Parameter	Description
pstIR	IR handle, should be IR0 / IR1 / IR2 / IR3.
punIrData	Configure data, refer to the unIR_SendSignalData_t enumeration definition.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

IR Decode HW APIs

rom_hw_timer_enable_decode

Function Prototype

```
EN_ERR_STA_T rom_hw_timer_enable_decode(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh);
```

Description

Enable the timer decode function.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which timer will be enabled, refer to the EN_TIMER_CH_T enumeration definition.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_timer_disable_decode

Function Prototype

```
EN_ERR_STA_T rom_hw_timer_disable_decode(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh);
```

Description

Disable the timer decode function.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which timer will be enabled, refer to the EN_TIMER_CH_T enumeration definition.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_timer_set_decode_pol

Function Prototype

```
EN_ERR_STA_T rom_hw_timer_set_decode_pol(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, EN_DECODE_POL_T enPol);
```

Description

Configure the decode polarity.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which timer will be enabled, refer to the EN_TIMER_CH_T enumeration definition.
enMode	Decode polarity, refer to the EN_DECODE_POL_T enumeration definition.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_timer_set_decode_mode

Function Prototype

```
EN_ERR_STA_T rom_hw_timer_set_decode_mode(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, EN_DECODE_MODE_T enMode);
```

Description

Configure the decode mode.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which timer will be enabled, refer to the EN_TIMER_CH_T enumeration definition.
enMode	Decode mode, refer to the EN_DECODE_MODE_T enumeration definition.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hw_timer_set_decode_interval

Function Prototype

```
EN_ERR_STA_T rom_hw_timer_set_decode_interval(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, uint8_t u8Val);
```

Description

Set an indicated timer decode interval.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which timer will be enabled, refer to the EN_TIMER_CH_T enumeration definition.
u8Val	The value of decode interval.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

IR Decode HAL APIs

rom_hal_ir_study_init

Function Prototype

```
EN_ERR_STA_T rom_hal_ir_study_init(stTIMER_Handle_t* pstIR, stDecodeInit_t* pstDecodeInit, stCapInit_t* pstCapInit);
```

Description

Initialize IR study.

Parameter

Parameter	Description
pstIR	IR handle, should be IR0 / IR1 / IR2 / IR3.
pstDecodeInit	Decode init struct type. .u8Prescale: The divider for the input clock, the range of u8Prescale is 0 ~ 15. .u8Mode: Decode mode, refer to the EN_DECODE_MODE_T enumeration definition. .u8Pol: Decode polarity, refer to the EN_DECODE_POL_T enumeration definition. .u8Value: The value of decode interval.
pstCapInit	Capture init struct type. .u8Prescale: The divider for the input clock, the range of u8Prescale is 0~15. .u8Signal: Capture signal, @ ref EN_CAP_SIGNAL_T. .u8Mode: Capture mode, @ ref EN_CAP_MODE_T.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hal_ir_start_study

Function Prototype

```
EN_ERR_STA_T rom_hal_ir_start_study(stTIMER_Handle_t* pstIR, uint8_t u8Rtune);
```

Description

Start IR study.

Parameter

Parameter	Description
pstIR	IR handle, should be IR0 / IR1 / IR2 / IR3.
u8Rtune	Set resistance of R in RC filter, $R = 50K \times 2^{(u8Rtune)}$.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hal_ir_stop_study

Function Prototype

EN_ERR_STA_T rom_hal_ir_stop_study(stTIMER_Handle_t* pstIR);

Description

Stop IR study.

Parameter

Parameter	Description
pstIR	IR handle, should be IR0 / IR1 / IR2 / IR3.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hal_timer_decode_init

Function Prototype

EN_ERR_STA_T rom_hal_timer_decode_init(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh, stDecodeInit_t* pstDecodeInit);

Description

Initialize an indicated timer, and configure decode.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which timer will be configured, refer to EN_TIMER_DECODE_CH_T enumeration definition.
pstDecodeInit	Decode init struct type. .u8Prescale: The divider for the input clock. 16-bit timer: The range of u8Prescale is [0:16). 32-bit timer: The range of u8Prescale is [0:256). .u8Mode: Decode mode, refer to the EN_DECODE_MODE_T. .u8Pol: Decode polarity, refer to the EN_DECODE_POL_T. .u8Value: The value of decode interval.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hal_timer_enable_decode

Function Prototype

```
EN_ERR_STA_T rom_hal_timer_enable_decode(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh);
```

Description

Enable the timer decode function.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which timer will be configured, refer to EN_TIMER_DECODE_CH_T enumeration definition.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

rom_hal_timer_disable_decode

Function Prototype

```
EN_ERR_STA_T rom_hal_timer_disable_decode(stTIMER_Handle_t* pstTIMER, EN_TIMER_CH_T enCh);
```

Description

Disable the timer decode function.

Parameter

Parameter	Description
pstTIMER	TIMER handle, should be TIMER0 / TIMER1 / TIMER2 / TIMER3.
enCh	Indicate which timer will be configured, refer to EN_TIMER_DECODE_CH_T enumeration definition.

Return

EN_ERR_STA_T	The function returns the status, refer to the EN_ERR_STA_T enumeration definition.
--------------	--

IR Module Examples

IR NEC Encode Example

IR NEC Encode Description

The IR module in the device supports a variety of protocols. This example uses the NEC infrared protocol as an example to introduce how to use the IR module.

In the NEC protocol, the frequency of the carrier is 38 kHz, and the values of the leader code, code 0, code 1, and repetition code are as follows:

Lead Code

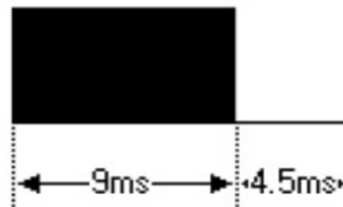


Figure 14. Lead Code

Bit Description

Bit 0: pulse time – 560 μ s, interval time – 565 μ s, period time – 1125 μ s

Bit 1: pulse time – 560 μ s, interval time – 1690 μ s, period time – 2250 μ s

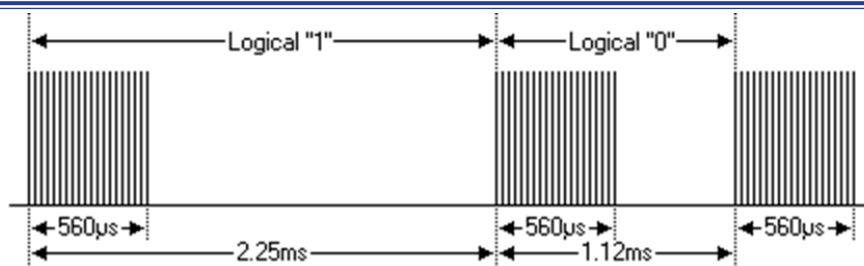


Figure 15. Bit Description

Repetition Code

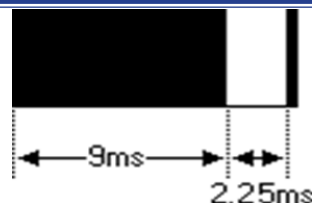


Figure 16. Repetition Code

IR NEC Encode Example Code

```
static uint8_t ir_tx_data_signal_generate (void)
{
    // GPIO init
    patch_hw_gpio_set_pin_input_output(GPIO_PORT_IR_OUT, GPIO_PIN_0,
    GPIO_MODE_IMPEDANCE);
    patch_hw_gpio_set_pin_pull_mode(GPIO_PORT_IR_OUT, GPIO_PIN_0, GPIO_PULL_
    NONE);
    patch_hw_gpio_set_pin_pid (GPIO_PORT_IR_OUT, GPIO_PIN_7, PID_IR_OUT);
    patch_hw_gpio_set_pin_pid(GPIO_PORT_PWM_OUT, GPIO_PIN_9, PID_GTIM_PWM0_CHA);
    patch_hw_gpio_set_pin_pid (GPIO_PORT_IR_SIGNLE_OUT, GPIO_PIN_11,
    PID_GTIM_PWM0_CHB);
```

```
// IR send init, config the leader code
rom_hal_ir_send_init(IR0, 38000, 50);
// Config data
rom_hw_ir_set_pwm_current_compare_and_polarity(IR0, CLEAR_SIGN(u16Data[0]),
PWM_POL_FALLING);
// Start send
rom_hal_start_send(IR0);
}
```

IR Decode Example

IR Decode Example Code

```
static uint8_t decode_func(void)
{
    // Decode Init struct type
    stDecodeInit_t g_stDecodeInit;
    g_stDecodeInit.u16CompVal = u32Compare;
    g_stDecodeInit.u8Prescale = u8Prescale;
    g_stDecodeInit.u8Mode = u8TrigMode;
    g_stDecodeInit.u8Pol = u8OutPol;
    g_stDecodeInit.u8Interval = u8Interval;
    g_stDecodeInit.u8Siganl = u8SignalSrc;
    // Decode GPIO Init
    patch_hw_gpio_set_pin_pull_mode(GPIOA, GPIO_PIN_0, GPIO_PULL_NONE);
    patch_hw_gpio_set_pin_input_output(GPIOA, GPIO_PIN_0, GPIO_MODE_IMPEDANCE);
    patch_hw_gpio_set_pin_pull_mode(GPIOA, GPIO_PIN_1, GPIO_PULL_NONE);
    patch_hw_gpio_set_pin_pull_mode(GPIOA, GPIO_PIN_2, GPIO_PULL_NONE);
    // Enable ir rx amp.
    patch_hw_gpio_enable_ir_rx_amp();
    // Set IR Amplifier amplification factor
    patch_hw_gpio_set_ir_rx_rtune_amp(1);
    // Set indicated pin peripheral function
    patch_hw_gpio_set_pin_pid(GPIO_PORT_TIM0_CHB, GPIO_PIN_TIM0_CHB,
PID_GTIM_DECODE0_CHB);
    // Ir function enable
    rom_hw_ir_enable(stTimerTestCfg.stTIMER_Handle);
    // Init a indicated timer, config decode.
    rom_hal_timer_decode_init(g_stTimerTestCfg.stTIMER_Handle, (EN_DECODE_CH_T)
g_stTimerTestCfg.EN_TIMER_CH, &g_stDecodeInit);
    // Enable timer decode function.
    rom_hal_timer_enable_decode(TIMERO, (EN_DECODE_CH_T)TIMER_CHB);
}
```

8 2nd-Boot

The device has three types of memory: ROM, Flash and SRAM. The ROM is used to store code and data, which cannot be modified; the Flash is used to store code and data, which can be modified by users. The SRAM is used to store running code and running data. The device memory map is as follows:

Table 5. Memory Map

Memory Type	Start Address	End Address
SRAM	0x20000000	0x2003FFFF
Flash	0x10000000	0x100FFFFFF
ROM	0x00000000	0x0003FFFF

Memory Map

After the device is powered on, the CPU starts from 0x00000000 (in ROM) by default. Since the ROM code cannot be modified, we designed the code which to load the boot code in the ROM to help the CPU jump to the SDK properly. This boot code is called 2nd-boot code.

The 2nd-boot code is stored in Flash and loaded in SRAM at runtime. 2nd-boot code is the link between ROM code and SDK. The device code running process as follows:

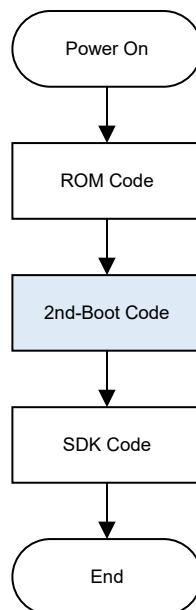


Figure 17. Start-up Flow

The 2nd-Boot Code

The 2nd-boot code has two main functions. First, determine whether there is a new version of application layer code in the OTA Code area, and do subsequent processing. The second is to initialize and enable cache so that code running in the Flash can be loaded into the cache. 2nd-boot code running process as follows:

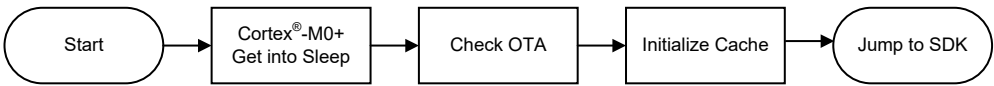


Figure 18. The 2nd-Boot Code Flow

OTA Flow

We divide Flash into four areas include 2nd-boot code area, OTA code information area, application code area and OTA code area (Sizes of each area can be modified by the user), as follows:

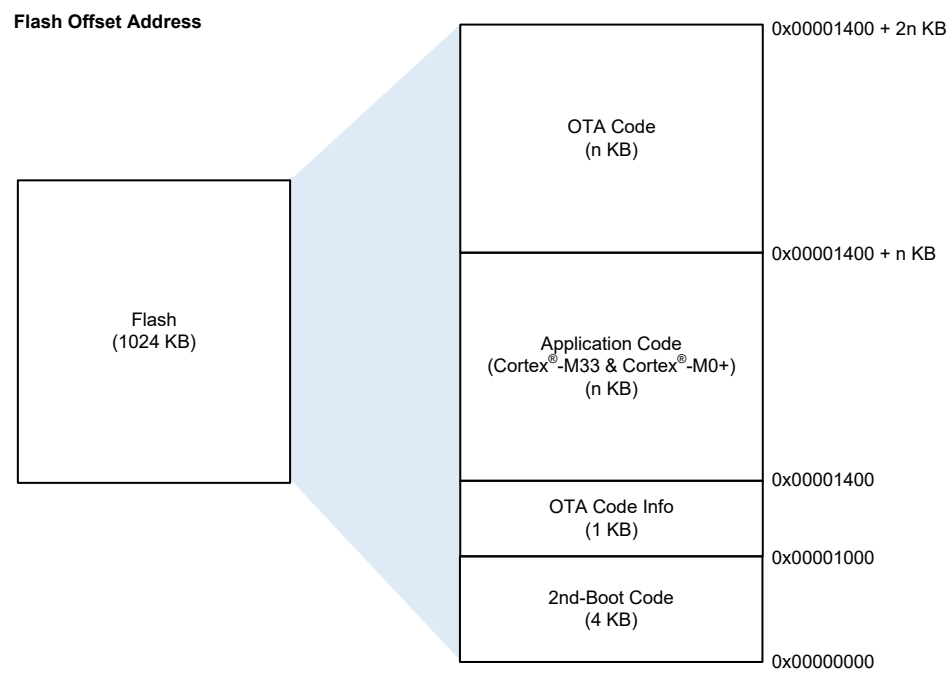


Figure 19. Flash Memory Map

The 2nd-boot code includes 2nd-boot code; the OTA code information includes running code information, OTA code size and CRC; the application code includes Cortex®-M33 and Cortex®-M0+ application code; the OTA code includes the code to be upgraded.

The OTA process in 2nd-boot includes 5 main steps as follows:

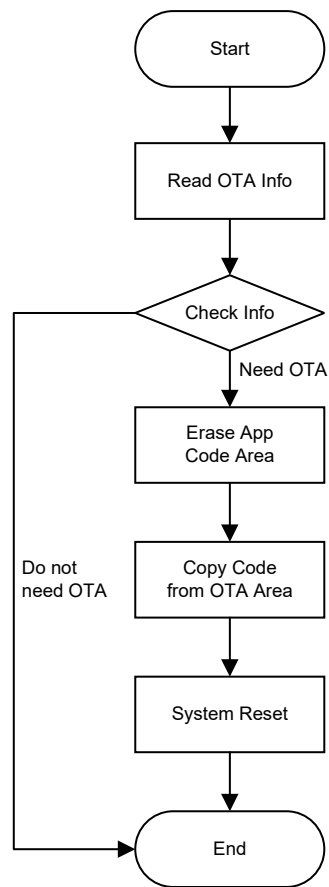


Figure 20. OTA Process in 2nd-Boot

Step 1. Read OTA information from Flash to check whether the upgrade is required.

If no, go to the next step to initialize cache;

If yes, we will step by step check ROM and 2nd-boot code version, OTA code CRC.

Step 2. Erase the application code to prepare copy OTA code to application code area.

Step 3. Read OTA code from OTA code area in Flash, and then write it to application code area.

Step 4. Check CRC of OTA code and CRC of new application code.

Step 5. If they are equal, modify the OTA information and reset the system. If they are not equal, retry from step 2.

Initialize Cache Read

Initializing cache read includes set cache read mode to QSPI, and enable cache read.

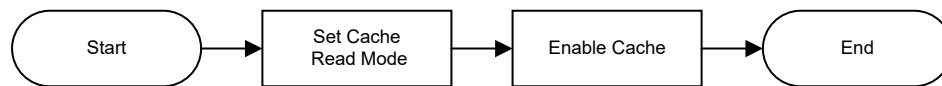


Figure 21. Initialize Cache

Jump to SDK

After the 2nd-code execution is completed, the code will jump to the SDK. We will modify PC pointer to reset handler of SDK.

9 IPC

Introduction

The device is a dual-core MCU consisting of a main processor, MP (Cortex®-M33), and a coprocessor, CP (Cortex®-M0+). The MP has powerful performance and can be used in applications that require powerful performance requirements, while the CP is mainly used for low power consumption and communication. During the use of dual cores, the MP and the CP often interact with each other in a way called Inter-Processor Communication (IPC). The underlying implementation of inter-processor communication is realized by the register-level mutual exclusion function, which ensures the multi-core data interaction security. To provide users with a clearer understanding of the IPC communication usage, the related APIs are described in detail below.

IPC API Introduction

IPC APIs Based on Message Queue Implementation

ipc_general_init

Function Prototype

```
EN_ERR_STA_T ipc_general_init(stIpcInit_t *pstIpcInit);
```

Functional Description

Initialize the IPC.

Parameter

Parameter		Description
pstIpcInit	pu8Buf	Pointer to the dual-core data shared buffer.
	u32BufSize	Buffer size shared by dual-core data.
	u32MsgCount	Maximum number of messages supported by a queue.
	u32MsgSize	Message size.
	enMode	QUEUE_MODE_OVERFLOW will be rewritten from the oldest data when its data is full. QUEUE_MODE_NO_OVERFLOW will reject new messages when its data is full.
	pfnCallback	A callback function called when another core sends a message.

Return Value

Status	Refer to the EN_ERR_STA_T.
--------	----------------------------

ipc_general_send_msg_blocking

Function Prototype

```
EN_ERR_STA_T ipc_general_send_msg_blocking(uint8_t *pu8Data, uint32_t u32DataLen);
```

Functional Description

Send messages in the blocking mode.

Parameter

Parameter	Description
pu8Data	Pointer to the data to be sent.
u32DataLen	Data size to be sent.

Return Value

Status	Refer to the EN_ERR_STA_T.
--------	----------------------------

ipc_general_send_msg_nonblocking

Function Prototype

```
EN_ERR_STA_T ipc_general_send_msg_nonblocking(uint8_t *pu8Data, uint32_t u32DataLen);
```

Functional Description

Send messages in the non-blocking mode.

Parameter

Parameter	Description
pu8Data	Pointer to the data to be sent.
u32DataLen	Data size to be sent.

Return Value

Status	Refer to the EN_ERR_STA_T.
--------	----------------------------

IPC APIs Based on Array Implementation

ipc_general_init

Function Prototype

```
EN_ERR_STA_T ipc_general_init(uint8_t *pu8DataBuf, uint32_t u32BufMaxLen, ipc_rx_msg_cb_func pfnCallback);
```

Functional Description

Initialize the IPC.

Parameter

Parameter	Description
pu8DataBuf	Pointer to the dual-core data shared buffer.
u32BufMaxLen	Buffer size shared by dual-core data.
pfnCallback	A callback function called when another core sends a message.

Return Value

Status	Refer to the EN_ERR_STA_T.
--------	----------------------------

ipc_general_send_msg_blocking

Function Prototype

```
uint32_t ipc_general_send_msg_blocking(uint8_t *pu8Data, uint32_t u32DataLen);
```

Functional Description

Send messages in the blocking mode.

Parameter

Parameter	Description
pu8Data	Pointer to the data to be sent.
u32DataLen	Data size to be sent.

Return Value

Status	Refer to the EN_ERR_STA_T.
--------	----------------------------

IPC Example Code

The MP and CP interact with each other using the IPC APIs.

IPC Initialization (MP is the same as CP)

```
// IPC communication buffer, can only be used for IPC communication
static uint8_t gu8IpcRxBuf[IPC_MSG_DATA_LEN + 4] __attribute__((aligned(4))) = {0};
// IPC initialization
ipc_general_init(gu8IpcRxBuf, sizeof(gu8IpcRxBuf), ipc_receive_msg_callback);
```

IPC Message Receiving Callback Function

```
// MP IPC callback function
static uint32_t ipc_receive_msg_callback(uint8_t *pu8Data, uint32_t u32DataLen)
{
    PRINTF("IPC MP Recv:");
    for(int i=0; i<u32DataLen;i++)
    {
        PRINTF("0x%02X", pu8Data[i]);
    } PRINTF("\r\n");
    return 0;
}

// CP IPC callback function
static uint32_t ipc_receive_msg_callback(uint8_t *pu8Data, uint32_t u32DataLen)
{
    PRINTF("IPC CP Recv:");
    for(int i=0; i<u32DataLen;i++)
    {
        PRINTF("0x%02X", pu8Data[i]);
    } PRINTF("\r\n");
    return 0;
}
```

IPC Message Sending Function

```
// MP sends messages to CP via IPC
for (i = 0; i < sizeof(gu8IpcTxBuf) - 1; i++)
{
    gu8IpcTxBuf[i] = rom_get_rand(); u8Checksum += gu8IpcTxBuf[i];
}
```

```
gu8IpcTxBuf[i] = u8Checksum;
PRINTF("IPC MP Send:");
for(int i=0; i<sizeof(gu8IpcTxBuf);i++)
{
    PRINTF("0x%02X",gu8IpcTxBuf[i]);
} PRINTF("\r\n");
ipc_general_send_msg_blocking(gu8IpcTxBuf, sizeof(gu8IpcTxBuf));
// CP sends messages to MP via IPC
for (i = 0; i < sizeof(gu8IpcTxBuf) - 1; i++)
{
    gu8IpcTxBuf[i] = rom_get_rand(); u8Checksum += gu8IpcTxBuf[i];
}
gu8IpcTxBuf[i] = u8Checksum;
PRINTF("IPC CP Send:");
for(int i=0; i<sizeof(gu8IpcTxBuf);i++)
{
    PRINTF("0x%02X",gu8IpcTxBuf[i]);
} PRINTF("\r\n");
ipc_general_send_msg_blocking(gu8IpcTxBuf, sizeof(gu8IpcTxBuf));
```

10 Bluetooth Low Energy GAP

Introduction

The Bluetooth® Low Energy GAP ensures that different Bluetooth products can discover each other and establish connections. The GAP defines how a Bluetooth device discovers and establishes secure/insecure connections to other devices. It handles some general mode services (such as inquiring, naming and searching) and some security issues (such as guarantees), as well as some connection related services (such as link establishment, channel and connection establishment). For this Bluetooth Low Energy device, the Bluetooth protocol stack part is located in the ROM. Through this document, users will become familiar with how to access the Bluetooth protocol stack using the standard APIs and be able to use the Bluetooth function proficiently.

Bluetooth Low Energy GAP API Interfaces

General Interfaces

rom_gap_api_set_random_device_address

Function Prototype

```
bool rom_gap_api_set_random_device_address(uint8_t pu8Address[6]);
```

Functional Description

Set a random device address.

Parameter

Parameter	Description
pu8Address[6]	Random address to be set.

Return Value

true or false	Indicate that sending a message to the underlying task succeeds or fails.
---------------	---

Note

Asynchronous interface, needs to send a message to the underlying task to execute the specific function. The execution result is obtained by event reporting.
If the device is currently in the advertising, scanning or initiating state, this API cannot be executed.

rom_gap_api_set_public_device_address

Function Prototype

```
bool rom_gap_api_get_public_device_address(uint8_t pu8Address[6]);
```

Functional Description

Set a public device address.

Parameter

Parameter	Description
pu8Address[6]	Public address to be set.

Return Value

true or false	Indicate that sending a message to the underlying task succeeds or fails.
---------------	---

Note

Asynchronous interface, needs to send a message to the underlying task to execute the specific function. The execution result is obtained by event reporting.
If the device is currently in the advertising, scanning or initiating state, this API cannot be executed.

rom_gap_api_get_random_device_address

Function Prototype

```
bool rom_gap_api_get_random_device_address(uint8_t pu8Address[6]);
```

Functional Description

Obtain the random device address.

Parameter

Parameter	Description
pu8Address[6]	Pointer to save the obtained random address.

Return Value

true or false	Indicate that the obtaining succeeds or fails.
---------------	--

Note

Synchronous interface, the random address is obtained after this function is executed.

rom_gap_api_get_public_device_address

Function Prototype

```
bool rom_gap_api_get_public_device_address(uint8_t pu8Address[6]);
```

Functional Description

Obtain the public device address.

Parameter

Parameter	Description
pu8Address[6]	Pointer to save the obtained public address.

Return Value

true or false	Indicate that the obtaining succeeds or fails.
---------------	--

Note

Synchronous interface, the public address is obtained after this function is executed.

rom_gap_api_clear_white_list

Function Prototype

```
bool rom_gap_api_clear_white_list(void);
```

Functional Description

Clear the white list.

Parameter

None.

Return Value

true or false	Indicate that sending a message to the underlying task succeeds or fails.
---------------	---

Note

Asynchronous interface, needs to send a message to the underlying task to execute the specific function. The execution result is obtained by event reporting.

rom_gap_api_add_device_to_white_list

Function Prototype

```
bool rom_gap_api_add_device_to_white_list(EN_GAP_WHITE_LIST_ADDRESS_TYPE_T
enumDeviceType, uint8_t pu8DeviceAddress[6]);
```

Functional Description

Add a device to the white list.

Parameter

Parameter	Description
enumDeviceType	Device address type to be added.
pu8Address[6]	Device address to be added.

Return Value

true or false	Indicate that sending a message to the underlying task succeeds or fails.
---------------	---

Note

Asynchronous interface, needs to send a message to the underlying task to execute the specific function. The execution result is obtained by event reporting.

rom_gap_api_remove_device_from_white_list

Function Prototype

```
bool rom_gap_api_remove_device_from_white_list(EN_GAP_WHITE_LIST_ADDRESS_TYPE_T
enumDeviceType, uint8_t pu8DeviceAddress[6]);
```

Functional Description

Remove a device from the white list.

Parameter

Parameter	Description
enumDeviceType	Device address type to be removed.
pu8Address[6]	Device address to be removed.

Return Value

true or false	Indicate that sending a message to the underlying task succeeds or fails.
---------------	---

Note

Asynchronous interface, needs to send a message to the underlying task to execute the specific function. The execution result is obtained by event reporting.

Advertising Interfaces

rom_gap_api_set_advertising_parameters

Function Prototype

```
bool rom_gap_api_set_advertising_parameters(uint8_t u8AdvertisingIndex,
stGapSetAdvertisingParameters_t* pstParam);
```

Functional Description

Set the advertising parameters.

Parameter

Parameter	Description
u8AdvertisingIndex	Advertising instance index.
pstParam	Advertising parameters.

Return Value

true or false	Indicate that sending a message to the underlying task succeeds or fails.
---------------	---

Note

Asynchronous interface, needs to send a message to the underlying task to execute the specific function. The execution result is obtained by event reporting.

Advertising parameter structure:

```
typedef struct __attribute__((packed))
{
    uint16_t u16AdvertisingIntervalMin625us;
    uint16_t u16AdvertisingIntervalMax625us;
    EN_GAP_ADVERTISING_TYPE_T enumAdvertisingType;
    EN_GAP_OWN_ADDRESS_TYPE_T enumOwnAddressType;
    EN_GAP_ADV_PEER_ADDRESS_TYPE_T enumPeerAddressType;
    uint8_t pu8PeerAddress[6];
    uint8_t u8AdvertisingChannelMap;
    EN_GAP_ADV_FILTER_POLICY_T enumAdvertisingFilterPolicy;
} stGapSetAdvertisingParameters_t;
```

u16AdvertisingIntervalMin625us	Minimum advertising interval, unit: 625 μ s.
u16AdvertisingIntervalMax625us	Maximum advertising interval, unit: 625 μ s.
enumAdvertisingType	Advertising types, which are connectable scannable, high duty cycle directed, scannable, non-connectable or non-scannable, and low duty cycle directed. GAP_ADVERTISING_TYPE_ADV_IND = 0x00, GAP_ADVERTISING_TYPE_DIRECT_IND_HIGH_DUTY_CYCLE = 0x01, GAP_ADVERTISING_TYPE_ADV_SCAN_IND = 0x02, GAP_ADVERTISING_TYPE_NONCONN_IND = 0x03, GAP_ADVERTISING_TYPE_DIRECT_IND_LOW_DUTY_CYCLE = 0x04.

enumOwnAddressType	Local device advertising address types, four types in total, the last two of which are resolvable private addresses: GAP_OWN_ADDRESS_TYPE_PUBLIC = 0x00, GAP_OWN_ADDRESS_TYPE_RANDOM = 0x01, GAP_OWN_ADDRESS_TYPE_RESOLVABLE_OR_PUBLIC = 0x02, GAP_OWN_ADDRESS_TYPE_RESOLVABLE_OR_RANDOM = 0x03.
enumPeerAddressType	Peer device advertising address types, two types in total. This parameter is valid when the directed advertising is used or when the local device address type is 2 or 3. GAP_ADV_PEER_ADDRESS_TYPE_PUBLIC_OR_PUBLIC_IDENTITY = 0x00, GAP_ADV_PEER_ADDRESS_TYPE_RANDOM_OR_RANDOM_IDENTITY = 0x01.
pu8PeerAddress[6]	Peer device advertising address. This parameter is valid when the directed advertising is used or when the local device address type is 2 or 3.
u8AdvertisingChannelMap	Advertising channel, set by bit: GAP_ADV_CHANNEL_MAP_37 = (1U << 0), GAP_ADV_CHANNEL_MAP_38 = (1U << 1), GAP_ADV_CHANNEL_MAP_39 = (1U << 2).
u8AdvertisingChannelMap	Advertising channel, set by bit: GAP_ADV_CHANNEL_MAP_37 = (1U << 0), GAP_ADV_CHANNEL_MAP_38 = (1U << 1), GAP_ADV_CHANNEL_MAP_39 = (1U << 2).
enumAdvertisingFilterPolicy	Advertising filtering rules: GAP_ADV_FILTER_POLICY_CONN_ALL_SCAN_ALL = 0x00, GAP_ADV_FILTER_POLICY_CONN_ALL_SCAN_WHITELIST = 0x01, GAP_ADV_FILTER_POLICY_CONN_WHITELIST_SCAN_ALL = 0x02, GAP_ADV_FILTER_POLICY_CONN_WHITELIST_SCAN_WHITELIST = 0x03, GAP_OWN_ADDRESS_TYPE_RESOLVABLE_OR_RANDOM = 0x03.

enumPeerAddressType	Peer device advertising address types, two types in total. This parameter is valid when the directed advertising is used or when the local device address type is 2 or 3. GAP_ADV_PEER_ADDRESS_TYPE_PUBLIC_OR_PUBLIC_IDENTITY = 0x00, GAP_ADV_PEER_ADDRESS_TYPE_RANDOM_OR_RANDOM_IDENTITY = 0x01,
pu8PeerAddress[6]	Peer device advertising address. This parameter is valid when the directed advertising is used or when the local device address type is 2 or 3.
u8AdvertisingChannelMap	Advertising channel, set by bit: GAP_ADV_CHANNEL_MAP_37 = (1U << 0), GAP_ADV_CHANNEL_MAP_38 = (1U << 1), GAP_ADV_CHANNEL_MAP_39 = (1U << 2).
enumAdvertisingFilterPolicy	Advertising filtering rules: GAP_ADV_FILTER_POLICY_CONN_ALL_SCAN_ALL = 0x00, GAP_ADV_FILTER_POLICY_CONN_ALL_SCAN_WHITELIST = 0x01, GAP_ADV_FILTER_POLICY_CONN_WHITELIST_SCAN_ALL = 0x02, GAP_ADV_FILTER_POLICY_CONN_WHITELIST_SCAN_WHITELIST = 0x03.

rom_gap_api_set_advertising_data

Function Prototype

```
bool rom_gap_api_set_advertising_data(uint8_t u8AdvertisingIndex, uint8_t* pu8AdvertisingData,
uint8_t u8AdvertisingDataLength);
```

Functional Description

Set the advertising data.

Parameter

Parameter	Description
u8AdvertisingIndex	Advertising instance index.
pu8AdvertisingData	Advertising data.
u8AdvertisingDataLength	Advertising data length.

Return Value

true or false	Indicate that sending a message to the underlying task succeeds or fails.
---------------	---

Note

Asynchronous interface, needs to send a message to the underlying task to execute the specific function. The execution result is obtained by event reporting.

rom_gap_api_set_scan_response_data

Function Prototype

```
bool rom_gap_api_set_scan_response_data(uint8_t u8AdvertisingIndex, uint8_t*
pu8ScanResponseData, uint8_t u8ScanResponseDataLength);
```

Functional Description

Set the advertising scan response data.

Parameter

Parameter	Description
u8AdvertisingIndex	Advertising instance index.
pu8ScanResponseData	Advertising scan response data.
u8ScanResponseDataLength	Advertising scan response data length.

Return Value

true or false	Indicate that sending a message to the underlying task succeeds or fails.
---------------	---

Note

Asynchronous interface, needs to send a message to the underlying task to execute the specific function. The execution result is obtained by event reporting.

rom_gap_api_set_advertising_enable

Function Prototype

```
bool rom_gap_api_set_advertising_enable(uint8_t u8AdvertisingIndex, bool bAdvertisingEnable);
```

Functional Description

Enable or disable the advertising.

Parameter

Parameter	Description
u8AdvertisingIndex	Advertising instance index.
bAdvertisingEnable	Enable or disable, true indicating enable, false indicating disable.

Return Value

true or false	Indicate that sending a message to the underlying task succeeds or fails.
---------------	---

Note

Asynchronous interface, needs to send a message to the underlying task to execute the specific function. The execution result is obtained by event reporting.

Scanning interfaces

rom_gap_api_set_scan_parameters

Function Prototype

```
bool rom_gap_api_set_scan_parameters(stGapSetScanParameters_t* pstParam);
```

Functional Description

Set the scan parameters.

Parameter

Parameter	Description
pstParam	Scan parameters. For details, refer to the following note description.

Return Value

true or false	Indicate that sending a message to the underlying task succeeds or fails.
---------------	---

Note

Asynchronous interface, needs to send a message to the underlying task to execute the specific function. The execution result is obtained by event reporting.

Scan parameter structure:

```
typedef struct __attribute__((packed))
{
    EN_GAP_SCAN_TYPE_T enumScanType; uint8_t u8ScanChannelMap;
    uint16_t u16ScanInterval625us;
    uint16_t u16ScanWindow625us;
    EN_GAP_OWN_ADDRESS_TYPE_T enumOwnAddressType;
    EN_GAP_SCAN_FILTER_POLICY_T enumScanningFilterPolicy;
} stGapSetScanParameters_t;
```

u16ScanInterval625us	Scan interval, unit: 625 μ s.
u16ScanWindow625us	Scan window, unit: 625 μ s.
enumScanType	Scan types, which are passive and active: GAP_SCAN_TYPE_PASSIVE = 0x00, GAP_SCAN_TYPE_ACTIVE = 0x01.
u8ScanChannelMap	Scan channel, set by bit: GAP_ADV_CHANNEL_MAP_37 = (1U << 0), GAP_ADV_CHANNEL_MAP_38 = (1U << 1), GAP_ADV_CHANNEL_MAP_39 = (1U << 2).
enumOwnAddressType	Local device address types, four types in total, the last two of which are resolvable private addresses: GAP_OWN_ADDRESS_TYPE_PUBLIC = 0x00, GAP_OWN_ADDRESS_TYPE_RANDOM = 0x01, GAP_OWN_ADDRESS_TYPE_RESOLVABLE_OR_PUBLIC = 0x02, GAP_OWN_ADDRESS_TYPE_RESOLVABLE_OR_RANDOM = 0x03.
enumScanningFilterPolicy	Scanning filtering rules: GAP_SCAN_FILTER_POLICY_ACCEPT_ALL = 00, GAP_SCAN_FILTER_POLICY_ACCEPT_WHITELIST = 01, GAP_SCAN_FILTER_POLICY_ACCEPT_ALL_NOT_RESOLVED_DIRECTED = 02, GAP_SCAN_FILTER_POLICY_ACCEPT_WHITELIST_NOT_RESOLVED_DIRECTED = 03.

rom_gap_api_set_scan_enable

Function Prototype

```
bool rom_gap_api_set_scan_enable(bool bScanEnable, bool bFilterDuplicates);
```

Functional Description

Enable or disable the scanning.

Parameter

Parameter	Description
bScanEnable	Enable or disable, true indicating enable, false indicating disable.
bFilterDuplicates	Whether to enable filtering duplicate devices. If enabled, duplicate devices will not be reported.

Return Value

true or false	Indicate that sending a message to the underlying task succeeds or fails.
---------------	---

Note

Asynchronous interface, needs to send a message to the underlying task to execute the specific function. The execution result is obtained by event reporting.

Initiating State Interfaces

rom_gap_api_create_connection

Function Prototype

```
bool rom_gap_api_create_connection(stGapCreateConnection_t* pstParam);
```

Functional Description

Enter the initiating state to create a connection.

Parameter

Parameter	Description
pstParam	Connection creation parameters. For details, refer to the following note description.

Return Value

true or false	Indicate that sending a message to the underlying task succeeds or fails.
---------------	---

Note

Asynchronous interface, needs to send a message to the underlying task to execute the specific function. The execution result is obtained by event reporting.

Connection creation parameter structure:

```
typedef struct __attribute__((packed))
{
    uint8_t u8ScanChannelMap;
    uint16_t u16ScanInterval625us;
    uint16_t u16ScanWindow625us;
    EN_GAP_INITIATOR_FILTER_POLICY_T enumInitiatorFilterPolicy;
    EN_GAP_INITIATOR_PEER_ADDRESS_TYPE_T enumPeerAddressType;
    uint8_t pu8PeerAddress[6];
    EN_GAP_OWN_ADDRESS_TYPE_T enumOwnAddressType;
    uint16_t u16ConnIntervalMin1250us;
    uint16_t u16ConnIntervalMax1250us; uint16_t u16ConnLatency;
    uint16_t u16SupervisionTimeout10ms;
} stGapCreateConnection_t;
```

u8ScanChannelMap	Scan channel, set by bit. Bits 0, 1, 2 represent channels 37, 38, 39 respectively.
u16ScanInterval625us	Scan interval, unit: 625 μ s.
u16ScanWindow625us	Scan window, unit: 625 μ s.
enumInitiatorFilterPolicy	Initiating connection filtering rules. There are two rules: GAP_INITIATOR_FILTER_POLICY_WHITELIST_IS_NOT_USED = 0x00, GAP_INITIATOR_FILTER_POLICY_WHITELIST_IS_USED = 0x01.
enumPeerAddressType	Peer device advertising address types, four types in total, the last two of which are identity addresses. GAP_INITIATOR_PEER_ADDRESS_TYPE_PUBLIC = 0x00, GAP_INITIATOR_PEER_ADDRESS_TYPE_RANDOM = 0x01, GAP_INITIATOR_PEER_ADDRESS_TYPE_PUBLIC_IDENTITY = 0x02, GAP_INITIATOR_PEER_ADDRESS_TYPE_RANDOM_IDENTITY = 0x03.
pu8PeerAddress[6]	Peer device address.
enumOwnAddressType	Local device address types, four types in total, the last two of which are resolvable private addresses. GAP_OWN_ADDRESS_TYPE_PUBLIC = 0x00, GAP_OWN_ADDRESS_TYPE_RANDOM = 0x01, GAP_OWN_ADDRESS_TYPE_RESOLVABLE_OR_PUBLIC = 0x02, GAP_OWN_ADDRESS_TYPE_RESOLVABLE_OR_RANDOM = 0x03.
u16ConnIntervalMin1250us	Minimum connection interval, unit: 1.25 ms.
u16ConnIntervalMax1250us	Maximum connection interval, unit: 1.25 ms.
u16ConnLatency	Connection latency parameter.
u16SupervisionTimeout10ms	Connection timeout parameter, unit: 10 ms.

rom_gap_api_create_connection_cancel

Function Prototype

```
bool rom_gap_api_create_connection_cancel(void);
```

Functional Description

Cancel the connection creation and exit the initiating state.

Parameter

None.

Return Value

true or false	Indicate that sending a message to the underlying task succeeds or fails.
---------------	---

Note

Asynchronous interface, needs to send a message to the underlying task to execute the specific function. The execution result is obtained by event reporting.

Connection State Interfaces

rom_gap_api_connection_parameters_update

Function Prototype

```
bool rom_gap_api_connection_parameters_update(uint16_t u16ConnHandle,  
stGapConnectionUpdate_t* pstParam);
```

Functional Description

Initiate the connection parameter update.

Parameter

Parameter	Description
u16ConnHandle	Connection index.
pstParam	Connection update parameters. For details, refer to the following note description.

Return Value

true or false	Indicate that sending a message to the underlying task succeeds or fails.
---------------	---

Note

Asynchronous interface, needs to send a message to the underlying task to execute the specific function. The execution result is obtained by event reporting.

Connection update parameter structure:

```
typedef struct __attribute__((packed))  
{  
    uint16_t u16ConnIntervalMin1250us;  
    uint16_t u16ConnIntervalMax1250us;  
    uint16_t u16ConnLatency;  
    uint16_t u16SupervisionTimeout10ms;  
} stGapConnectionUpdate_t;
```

u16ConnIntervalMin1250us	Minimum connection interval, unit: 1.25 ms.
u16ConnIntervalMax1250us	Maximum connection interval, unit: 1.25 ms.
u16ConnLatency	Connection latency parameter.
u16SupervisionTimeout10ms	Connection timeout parameter, unit: 10 ms.

rom_gap_api_connection_set_phy

Function Prototype

```
bool rom_gap_api_connection_set_phy(uint16_t u16ConnHandle, stGapSetPhy_t* pstParam);
```

Functional Description

Set the receiver/transmitter PHY for the connection.

Parameter

Parameter	Description
u16ConnHandle	Connection index.
pstParam	Receiver/transmitter PHY parameters. For details, refer to the following note description.

Return Value

true or false	Indicate that sending a message to the underlying task succeeds or fails.
---------------	---

Note

Asynchronous interface, needs to send a message to the underlying task to execute the specific function.

The underlying Task negotiates with the peer device to set the receiver/transmitter PHY for the connection. The PHY information after taking effect is obtained by event reporting.

Receiver/transmitter PHY parameter structure:

```
typedef struct __attribute__((packed))
{
    uint8_t u8PreferTxPhys; uint8_t u8PreferRxPhys;
    EN_GAP_CODED_PHY_OPTION_T enumCodedPhyOption;
} stGapSetPhy_t;
```

u8PreferTxPhys	Preferred PHYs for transmitting data, three types in total: GAP_PREFER_PHY_1M = 0, GAP_PREFER_PHY_2M = 1, GAP_PREFER_PHY_CODED = 2.
u8PreferRxPhys	Preferred PHYs for receiving data, three types in total: GAP_PREFER_PHY_1M = 0, GAP_PREFER_PHY_2M = 1, GAP_PREFER_PHY_CODED = 2.
enumCodedPhyOption	If the previous parameter u8PreferTxPhys desires to use a coded PHY, which one is preferred, as shown below: GAP_CODED_PHY_OPTION_TX_NO_PREFER = 0, GAP_CODED_PHY_OPTION_TX_PREFER_S2 = 1, GAP_CODED_PHY_OPTION_TX_PREFER_S8 = 2.

rom_gap_api_read_remote_version

Function Prototype

```
bool rom_gap_api_read_remote_version(uint16_t u16ConnHandle);
```

Functional Description

Read the peer device version information.

Parameter

Parameter	Description
u16ConnHandle	Connection index.

Return Value

true or false	Indicate that sending a message to the underlying task succeeds or fails.
---------------	---

Note

Asynchronous interface, needs to send a message to the underlying task to execute the specific function.

The underlying Task sends a message to the peer device to obtain the version information. After receiving a reply from the peer device, the version information is obtained by event reporting.

rom_gap_api_read_remote_features

Function Prototype

```
bool rom_gap_api_read_remote_features(uint16_t u16ConnHandle);
```

Functional Description

Read the features supported by the peer device.

Parameter

Parameter	Description
u16ConnHandle	Connection index.

Return Value

true or false	Indicate that sending a message to the underlying task succeeds or fails.
---------------	---

Note

Asynchronous interface, needs to send a message to the underlying task to execute the specific function.

The underlying Task sends a message to the peer device to obtain its supported features. After receiving a reply from the peer device, the features supported by both sides is obtained by event reporting.

rom_gap_api_set_data_length

Function Prototype

```
bool rom_gap_api_set_data_length(uint16_t u16ConnHandle, uint16_t u16TxOctets);
```

Functional Description

Set the maximum PDU transmission packet length.

Parameter

Parameter	Description
u16ConnHandle	Connection index.
u16TxOctets	Maximum PDU transmission packet length.

Return Value

true or false	Indicate that sending a message to the underlying task succeeds or fails.
---------------	---

Note

Asynchronous interface, needs to send a message to the underlying task to execute the specific function.

The underlying Task negotiates the maximum transmission packet length with the peer device. The finally determined information is obtained by event reporting.

rom_gap_api_get_mtu

Function Prototype

```
uint16_t rom_gap_api_get_mtu(uint16_t u16ConnHandle);
```

Functional Description

Read the MTU for the current connection.

Parameter

Parameter	Description
u16ConnHandle	Connection index.

Return Value

uint16_t	MTU that is read out.
----------	-----------------------

Note

Synchronous interface.

rom_gap_api_read_rssi

Function Prototype

```
bool rom_gap_api_read_rssi(uint16_t u16ConnHandle);
```

Functional Description

Read the signal strength RSSI for the current connection.

Parameter

Parameter	Description
u16ConnHandle	Connection index.

Return Value

true or false	Indicate that sending a message to the underlying task succeeds or fails.
---------------	---

Note

Asynchronous interface, needs to send a message to the underlying task to execute the specific function. The execution result and RSSI value are obtained by event reporting.

rom_gap_api_update_channel_map

Function Prototype

```
bool rom_gap_api_update_channel_map(uint16_t u16ConnHandle, uint8_t pu8ChannelMap[5]);
```

Functional Description

Set the channel used for the connection.

Parameter

Parameter	Description
u16ConnHandle	Connection index.
pu8ChannelMap	Connection channel to be set.

Return Value

true or false	Indicate that sending a message to the underlying task succeeds or fails.
---------------	---

Note

Asynchronous interface, needs to send a message to the underlying task to execute the specific function. The execution result is obtained by event reporting.

rom_gap_api_read_channel_map

Function Prototype

```
bool rom_gap_api_read_channel_map(uint16_t u16ConnHandle);
```

Functional Description

Read the channel used for the current connection.

Parameter

Parameter	Description
u16ConnHandle	Connection index.

Return Value

true or false	Indicate that sending a message to the underlying task succeeds or fails.
---------------	---

Note

Asynchronous interface, needs to send a message to the underlying task to execute the specific function. The execution result and RSSI value are obtained by event reporting.

rom_gap_api_disconnect

Function Prototype

```
bool rom_gap_api_disconnect(uint16_t u16ConnHandle, EN_GAP_DISCONNECT_REASON_T  
enumDisconnectReason);
```

Functional Description

Disconnect the current connection.

Parameter

Parameter	Description
u16ConnHandle	Connection index.
enumDisconnectReason	Disconnection reason.

Return Value

true or false	Indicate that sending a message to the underlying task succeeds or fails.
---------------	---

Note

Asynchronous interface, needs to send a message to the underlying task to execute the specific function. The execution result and RSSI value are obtained by event reporting.

Pairing and Encrypted Connection Related Interfaces

rom_gap_api_update_pair_para

Function Prototype

```
void rom_gap_api_update_pair_para(uint8_t iocap, uint8_t authreq, uint8_t keysize);
```

Functional Description

Set the pairing related parameters.

Parameter

Parameter	Description
iocap	Device I/O capability, including input and output capability, ranging from 0 to 4, as follows: IO_CAPABILITY_DISPLAY_ONLY = 0, IO_CAPABILITY_DISPLAY_YES_NO = 1, IO_CAPABILITY_KEYBOARD_ONLY = 2, IO_CAPABILITY_NO_INPUT_NO_OUTPUT = 3, IO_CAPABILITY_KEYBOARD_DISPLAY = 4.
authreq	Pairing authentication requirement, Bit 0 ~ 3 valid, two or more bits can be combined: Bit 0: SM_AUTHREQ_BONDING Bit 1: SM_AUTHREQ_MITM_PROTECTION Bit 2: SM_AUTHREQ_SECURE_CONNECTION Bit 3: SM_AUTHREQ_KEYPRESS
Keysize	Minimum encryption key length, ranging from 7 to 16.

Return Value

None.

Note

Synchronous interface. It is usually called when the device is powered on and initialized. The parameters set here and the presence or absence of peer OOB data determine which pairing mode and pairing method are used.

rom_gap_api_register_oob_data_callback

Function Prototype

```
void rom_gap_api_register_oob_data_callback(int (*get_oob_data_callback) (uint8_t peer_addr_type, uint8_t peer_addr[6], uint8_t oob_data[16]));
```

Functional Description

Set the callback function for obtaining the peer device OOB data in the traditional pairing mode.

Parameter

Parameter	Description
int (*get_oob_data_callback)	Callback function for obtaining the peer device OOB data, called during pairing.

Return Value

None.

Note

Synchronous interface. It is usually called when the device is powered on and initialized. If this API is not called to set the callback function, it indicates that no peer device OOB data exists. The oob_data[16] parameter in the callback function is the peer device OOB data. In the traditional pairing mode, if this callback function is set, it indicates that the peer device OOB data exists. The OOB data is used as TK to generate Confirm data when pairing using the OOB pairing method (the method can be used only when both devices have their peer device OOB data).

rom_gap_api_sm_generate_local_sc_oob_data_callback

Function Prototype

```
void rom_gap_api_register_sc_oob_data_callback(int (*get_sc_oob_data_callback) (uint8_t address_type, uint8_t addr[6], uint8_t * oob_sc_peer_confirm, uint8_t * oob_sc_peer_random));
```

Functional Description

Set the callback function for obtaining the peer device OOB data in the secure pairing mode. The peer OOB data can be obtained from the peer device by using the rom_gap_api_sm_generate_local_sc_oob_data.

Parameter

Parameter	Description
int (*get_oob_data_callback)	Callback function for obtaining the peer device OOB data, called during pairing.

Return Value

None.

Note

Synchronous interface. It is usually called when the device is powered on and initialized. If this API is not called to set the callback function, it indicates that no peer device OOB data exists. The confirm_value and random_value parameters in the callback function are the peer device OOB data, which are the generated confirm value and the used random number respectively. The confirm value is generated by the random number and the public key. In the secure pairing mode, if this callback function is set, it indicates that the peer device OOB data exists. The confirm value and random number obtained by the callback function are used for authentication verification when pairing using the OOB pairing method (the method can be used as long as either one device has the peer device OOB data).

rom_gap_api_sm_generate_local_sc_oob_data

Function Prototype

```
void rom_gap_api_sm_generate_local_sc_oob_data (void (*callback) (const uint8_t *confirm_value, const uint8_t *random_value));
```

Functional Description

Generate the local OOB data in the secure pairing mode.

Parameter

Parameter	Description
void (*callback)	A callback function needs to be registered to receive the local OOB data: confirm value and random number. The confirm_value and random_value parameters in the callback function are these OOB data. The confirm value is generated by the random number and the public key.

Return Value

None.

Note

Synchronous interface. Only when the OOB data is sent to the peer device using the out-of-band (non-Bluetooth transmission) method, can the peer device have the local OOB data.

rom_gap_api_send_security_req

Function Prototype

```
void rom_gap_api_send_security_req (uint16_t u16ConnHandle);
```

Functional Description

For the Master device: Enable pairing, or enable encryption if already paired.

For the Slave device: Send a security request to the Master device to enable the Master device to pair or encrypt.

Parameter

Parameter	Description
u16ConnHandle	Connection index.

Return Value

None.

Note

Synchronous interface.

rom_gap_api_input_passkey_during_pair

Function Prototype

```
void rom_gap_api_input_passkey_during_pair(uint16_t u16ConnHandle, uint32_t u32Passkey);
```

Functional Description

Set a PIN code for pairing. During the pairing process, if the local device needs to input a PIN code, the input PIN code is passed to the protocol stack via this API to complete the subsequent pairing process.

Parameter

Parameter	Description
u16ConnHandle	Connection index.
u32Passkey	PIN code value to be input, ranging from 000000 to 999999.

Return Value

None.

Note

Synchronous interface.

rom_gap_api_sm_numeric_comparison_confirm

Function Prototype

```
void rom_gap_api_sm_numeric_comparison_confirm(uint16_t u16ConnHandle, uint8_t confirm_
success);
```

Functional Description

During secure pairing, both devices display a confirmation code respectively. The local device is required to compare whether its confirmation code is consistent with that of the peer device. The comparison result is sent to the protocol stack via this API.

Parameter

Parameter	Description
u16ConnHandle	Connection index.
confirm_success	1 indicates that the confirmation code of the local device is consistent with that of the peer device, while 0 indicates that it is not.

Return Value

None.

Note

Synchronous interface.

rom_gap_api_set_secure_connections_only

Function Prototype

```
void rom_gap_api_set_secure_connections_only (uint8_t u8Enable);
```

Functional Description

Set whether to use secure pairing mode only. If only secure pairing mode is used, pairing cannot be implemented if the secure pairing mode is not selected during pairing.

Parameter

Parameter	Description
u8Enable	1 indicates only secure pairing mode is used, while 0 indicates that it is not.

Return Value

None.

Note

Synchronous interface.

rom_gap_api_long_term_key_request_reply

Function Prototype

```
void rom_gap_api_long_term_key_request_reply(uint16_t u16ConnHandle, uint8_t* pu8Ltk);
```

Functional Description

Set an LTK for the encrypted connection, only available for the slave devices. After the device is paired with the peer device successfully, it is connected again. When the callback message for requesting LTK, MSG_LTK_REQ_WHEN_RECONNECT_AFTER_PAIR_IND, is received, the saved LTK is passed to the protocol stack via this API to complete the connection encryption process.

Parameter

Parameter	Description
u16ConnHandle	Connection index.
pu8Ltk	LTK for the encrypted connection.

Return Value

None.

Note

Synchronous interface.

rom_gap_api_long_term_key_request_negative_reply

Function Prototype

```
void rom_gap_api_long_term_key_request_negative_reply(uint16_t u16ConnHandle);
```

Functional Description

When the callback message for requesting LTK, MSG_LTK_REQ_WHEN_RECONNECT_AFTER_PAIR_IND, is received after connection, if the corresponding saved LTK cannot be found, this API is called to indicate that the connection cannot be encrypted without LTK. This is also only available for the slave devices.

Parameter

Parameter	Description
u16ConnHandle	Connection index.

Return Value

None.

Note

Synchronous interface.

rom_gap_api_set_pair_pin_code

Function Prototype

```
void rom_gap_api_set_pair_pin_code(bool is_use_random_num, uint32_t u32PinNum);
```

Functional Description

Set the method for the local device to generate a PIN code during pairing.

Parameter

Parameter	Description
is_use_random_num	true indicates the random generation method. When the local device is required to generate a PIN during each pairing, a random number is generated as the PIN, ignoring the second parameter. false Indicates the method of using the fixed PIN code. The fixed PIN code value is represented by the second parameter value.
u32PinNum	Fixed PIN code value.

Return Value

None.

Note

Synchronous interface.

rom_gap_api_sm_bond_info_save_by_app_config_only_for_legacy_pair

Function Prototype

```
void rom_gap_api_sm_bond_info_save_by_app_config_only_for_legacy_pair(uint32_t onoff);
```

Functional Description

In the traditional pairing mode, when the connection requests LTK again after pairing, the protocol stack can calculate the LTK from the carried RAND and EDIV information. Therefore, the pairing information can be saved without using an APP. This API is used to configure whether the pairing information is saved by the APP. If not, the protocol stack will automatically process and calculate the LTK to complete the encryption and the callback message for requesting LTK, MSG_LTK_REQ_WHEN_RECONNECT_AFTER_PAIR_IND, will not be sent when the next reconnection is encrypted. Otherwise, the pairing information is saved by the APP. The callback message for requesting LTK will be sent to the protocol stack to complete the connection encryption when the next reconnection is encrypted.

Parameter

Parameter	Description
onoff	0 indicates that the APP is not required to save the pairing information; otherwise, it indicates that it is required.

Return Value

None.

Note

Synchronous interface.

rom_gap_api_set_bond_info_for_setup_encryption

Function Prototype

```
void rom_gap_api_set_bond_info_for_setup_encryption (uint16_t u16ConnHandle, stGapBondInfo_t* pstBondinfo);
```

Functional Description

This API needs to be called when entering the connection to load the saved pairing information (including LTK, RAND, EDIV and other parameters required for encrypting the connection) to the protocol stack for the encryption process.

Parameter

Parameter	Description
u16ConnHandle	Connection index.
pstBondInfo	Pairing information. For details, refer to the following note description.

Return Value

None.

Note

Synchronous interface.

Pairing information structure:

```
typedef struct __attribute__((packed))
{
    uint16_t u16Ediv;
    uint8_t pu8Rand[8];
    uint8_t pu8Ltk[16];
    uint8_t u8KeySize;
    uint8_t u8Authenticated;
    uint8_t u8Authorized;
} stGapBondInfo_t;
```

u16Ediv	EDIV parameter.
pu8Rand[8]	Random number RAND parameter.
pu8Ltk[16]	Encryption key LTK.
u8KeySize	Effective key length.
u8Authenticated	Authenticated or not.
u8Authorized	Authorized or not.

Privacy Function Related Interfaces

rom_gap_api_add_device_to_resolving_list

Function Prototype

```
bool rom_gap_api_add_device_to_resolving_list(uint8_t Peer_Identity_Address_Type , uint8_t
*Peer_Identity_Address , uint8_t *Peer_IRK , uint8_t* Local_IRK);
```

Functional Description

Add a peer device to the private address resolving list.

Parameter

Parameter	Description
Peer_Identity_Address_Type	Peer device identity address type.
Peer_Identity_Address	Peer device identity address.
Peer_IRK	Peer device IRK.
Local_IRK	Local device IRK.

Return Value

true or false	Indicate that sending a message to the underlying task succeeds or fails.
---------------	---

Note

Asynchronous interface, needs to send a message to the underlying task to execute the specific function. The execution result and channel information are obtained by event reporting.

rom_gap_api_remove_device_from_resolving_list

Function Prototype

```
bool rom_gap_api_remove_device_from_resolving_list (uint8_t Peer_Identity_Address_Type ,  
uint8_t Peer_Identity_Address[6]);
```

Functional Description

Remove a device to the private address resolving list.

Parameter

Parameter	Description
Peer_Identity_Address_Type	Peer device identity address type to be removed.
Peer_Identity_Address	Peer device identity address to be removed.

Return Value

true or false	Indicate that sending a message to the underlying task succeeds or fails.
---------------	---

Note

Asynchronous interface, needs to send a message to the underlying task to execute the specific function. The execution result and channel information are obtained by event reporting.

rom_gap_api_clear_resolving_list

Function Prototype

```
bool rom_gap_api_clear_resolving_list (void);
```

Functional Description

Clear the private address resolving list.

Parameter

None.

Return Value

true or false	Indicate that sending a message to the underlying task succeeds or fails.
---------------	---

Note

Asynchronous interface, needs to send a message to the underlying task to execute the specific function. The execution result and channel information are obtained by event reporting.

gap_api_set_period_to_update_resolved_address

Function Prototype

```
void gap_api_set_period_to_update_resolved_address(uint32_t u32Ms);
```

Functional Description

Set the private address update period.

Parameter

Parameter	Description
u16Second	Private address update period, unit: second, ranging from 1 to 0xA1B8.

Return Value

None.

Note

Synchronous interface.

rom_gap_api_set_privacy_mode_for_peer_device

Function Prototype

```
bool rom_gap_api_set_privacy_mode_for_peer_device (uint8_t Peer_Identity_Address_Type ,  
uint8_t Peer_Identity_Address[6], uint8_t Privacy_Mode);
```

Functional Description

Set the privacy mode for the device in the resolving list.

Parameter

Parameter	Description
Peer_Identity_Address_Type	Peer device identity address type.
Peer_Identity_Address	Peer device identity address.
Privacy_Mode	Privacy mode. 0 indicates network privacy mode, 1 indicates device privacy mode, and other values indicate reserved.

Return Value

true or false	Indicate that sending a message to the underlying task succeeds or fails.
---------------	---

Note

Asynchronous interface, needs to send a message to the underlying task to execute the specific function. The execution result and channel information are obtained by event reporting.

rom_gap_api_set_addr_resolution_enable

Function Prototype

```
bool rom_gap_api_set_addr_resolution_enable (uint8_t enable);
```

Functional Description

Enable or disable the private address resolution.

Parameter

Parameter	Description
enable	0 indicates disable, 1 indicates enable, and other values indicate reserved.

Return Value

true or false	Indicate that sending a message to the underlying task succeeds or fails.
---------------	---

Note

Asynchronous interface, needs to send a message to the underlying task to execute the specific function. The execution result and channel information are obtained by event reporting.

Stack Messages

MSG_BLE_CONNECTED_IND

This is a message indicating that the connection has been established. This message is received when the connection has been successfully established. It contains the connection parameters and the peer device address information. Its data is a structure, as described below.

```
typedef struct __attribute__((packed))
{
    uint8_t Subevent_Code;
    uint8_t Status;
    uint16_t Connection_Handle;
    uint8_t Role;
    uint8_t Peer_Address_Type;
    uint8_t Peer_Address[6];
    uint8_t Peer_RPA_Resolved;
    uint8_t Peer_Identify_Address_Type;
    uint8_t Peer_Identify_Address[6];
    uint16_t Conn_Interval;
    uint16_t Conn_Latency;
    uint16_t Supervision_Timeout;
    uint8_t Master_Clock_Accuracy;
    uint8_t Own_Address_Type;
    uint8_t Own_Address_Be_RPA; uint8_t Own_Address[6];
} stHciEventParamVendorConnectionComplete_t;
```

Its main parameters are as follows:

Connection_Handle	Connection index.
Role	Connection role. 0 indicates master and 1 indicates slave.
Peer_Address_Type	Peer device address type. 0 indicates public address and 1 indicates random address.
Peer_Address[6]	Peer device address (in little-endian order). If it is a random address and the highest 2 bits are 01, the address is a resolvable private address RPA.
Peer_RPA_Resolved	Peer_Address successfully resolved or not (Prerequisite: Peer_Address is RPA).
Peer_Identify_Address_Type	Identity address type corresponding to the conditions where Peer_Address is RPA and the resolution is successful.
Peer_Identify_Address[6]	Identity address corresponding to the conditions where Peer_Address is RPA and the resolution is successful.
Conn_Interval	Connection interval, unit: 1.25 ms.
Conn_Latency	Connection latency parameter.
Supervision_Timeout	Connection timeout parameter, unit: 10 ms.
Master_Clock_Accuracy	Peer master device clock accuracy. This parameter is valid only when the local device is in the slave role.
Own_Address_Type	Local address type. 0 indicates public address and 1 indicates random address.
Own_Address_Be_RPA	Whether the local address is RPA.
Own_Address[6]	Local address.

MSG_BLE_DISCONNECTED_IND

This is a message indicating that the connection has been disconnected. This message is received when the connection has been disconnected. Its data is a structure, as described below.

```
typedef struct __attribute__((packed))
{
    uint8_t Status;
    uint16_t Connection_Handle;
    uint8_t Reason;
} stHciEventParamDisconnectionComplete_t;
```

Its main parameters are as follows:

Connection_Handle	Connection index.
Reason	Disconnection reason.

MSG_BLE_CONNECTION_UPDATE_COMPLETE_IND

This is a message indicating that the connection parameters have been updated. This message is reported when the underlying protocol stack has completed the connection parameter update. Its data is a structure, as described below.

```
typedef struct __attribute__((packed))
{
    uint8_t Subevent_Code;
    uint8_t Status;
    uint16_t Connection_Handle;
    uint16_t Conn_Interval;
    uint16_t Conn_Latency;
    uint16_t Supervision_Timeout;
} stHciEventParamLEConnectionUpdateComplete_t;
```

Its main parameters are as follows:

Connection_Handle	Connection index.
Conn_Interval	Connection interval, unit: 1.25 ms.
Conn_Latency	Connection latency parameter.
Supervision_Timeout	Connection supervision timeout period, unit: 10 ms.

MSG_BLE_LL_VERSION_IND

This is a message indicating the Bluetooth version number of the peer device. This message is reported when the underlying protocol stack has received the version number of the peer device. Its data is a structure, as described below.

```
typedef struct __attribute__((packed))
{
    uint8_t Status;
    uint16_t Connection_Handle;
    uint8_t Version;
    uint16_t Manufacturer_Name;
    uint16_t Subversion;
} stHciEventParamReadRemoteVersionInformationComplete_t;
```

Its main parameters are as follows:

Connection_Handle	Connection index.
Version	Link Layer version number, for example, 10 indicates version 5.1 and 11 indicates version 5.2.
Manufacturer_Name	Bluetooth chip manufacturer ID (ID number assigned by SIG to each manufacturer).
Subversion	Subversion number.

MSG_BLE_LL_FEATTRUE_IND

This is a message indicating the Bluetooth features of the peer device. This message is reported when the underlying protocol stack has obtained the Bluetooth features of the peer device. Its data is a structure, as described below.

```
typedef struct __attribute__((packed))
{
    uint8_t Subevent_Code;
    uint8_t Status;
    uint16_t Connection_Handle;
    uint64_t LE_Features;
} stHciEventParamLEReadRemoteFeaturesComplete_t;
```

Its main parameters are as follows:

Connection_Handle	Connection index.
LE_Features	Supported Bluetooth features, 64 bits in total, each bit indicates a supported feature.

MSG_BLE_DATA_LENGTH_UPDATE_COMPLETE_IND

This is a message indicating that the Link layer packet length update has completed. This message is reported when the underlying protocol stack has completed the Link layer packet length update. Its data is a structure, as described below.

```
typedef struct __attribute__((packed))
{
    uint8_t Subevent_Code;
    uint16_t Connection_Handle;
    uint16_t MaxTxOctets;
    uint16_t MaxTxTime;
    uint16_t MaxRxOctets;
    uint16_t MaxRxTime;
} stHciEventParamLEDataLengthChange_t;
```

Its main parameters are as follows:

Connection_Handle	Connection index.
MaxTxOctets	Maximum transmitted data packet length.
MaxTxTime	Maximum time for transmitting data packets, unit: μ s.
MaxRxOctets	Maximum received packet length.
MaxRxTime	Maximum time for receiving data packets, unit: μ s.

MSG_BLE_PHY_UPDATE_COMPLETE_IND

This is a message indicating that the PHY update has completed. This message is reported when the underlying protocol stack has completed the PHY update. Its data is a structure, as described below.

```
typedef struct __attribute__((packed))
{
    uint8_t Subevent_Code;
    uint8_t Status;
    uint16_t Connection_Handle;
    uint8_t TX_PHY;
    uint8_t RX_PHY;
} stHciEventParamLEPhyUpdateComplete_t;
```

Its main parameters are as follows:

Connection_Handle	Connection index.
TX_PHY	PHY used for transmitting data, can be 1, 2 and 3, which indicate 1M, 2M and LE Coded respectively.
RX_PHY	PHY used for receiving data, can be 1, 2 and 3, which indicate 1M, 2M and LE Coded respectively.

MSG_BLE_LLCP_CONN_UPDATE_RSP_IND

This indicates a response message received to the connection update request initiated by the Slave device at the LLCAP layer. This message mainly displays whether the peer device accepts the connection update request sent by the local device. If accepted, the peer device will initiate the connection parameter update procedure (MSG_BLE_CONNECTION_UPDATE_COMPLETE_IND will be reported when the procedure is completed). Otherwise, if rejected, the connection parameter update procedure will not be initiated. Its data is a structure, as described below.

```
typedef struct __attribute__((packed))
{
    uint16_t conn_handle;
    uint16_t result;
} st_llcap_conn_update_rsp_event;
```

Its main parameters are as follows:

Connection_Handle	Connection index.
result	0 indicates accepted, otherwise rejected.

MSG_BLE_MTU_EXCHANGED_IND

This is a message indicating that the MTU update has completed. This message is reported when the underlying protocol stack has completed the MTU update. Its data is a structure, as described below.

```
typedef struct __attribute__((packed))
{
    uint16_t conn_handle;
    uint16_t mtu;
} st_mtu_exchange_complete_event;
```

Its main parameters are as follows:

Connection_Handle	Connection index.
mtu	Updated MTU.

MSG_BLE_ENCRYPTED_CHANGED_IND

This is a message indicating that the encryption state for the connection has changed. Its data is a structure, as described below.

```
typedef struct __attribute__((packed))
{
    uint8_t Status;
    uint16_t Connection_Handle;
    uint8_t Encryption_Enabled;
} stHciEventParamEncryptionChange_t;
```

Its main parameters are as follows:

Status	0 indicates that the encryption state has changed. 6 indicates that the encryption fails due to an incorrect PIN code.
Connection_Handle	Connection index.
Encryption_Enabled	Changed encryption state. 1 indicates encrypted, and 0 indicates not encrypted.

MSG_BLE_ENCRYPTED_REFRESH_IND

This is a message indicating that the encryption key for the connection has been updated. This message is reported when the underlying protocol stack pauses encryption and then updates the key to re-enter the encrypted state. Its data is a structure, as described below.

```
typedef struct __attribute__((packed))
{
    uint8_t Status;
    uint16_t Connection_Handle;
} stHciEventParamEncryptionKeyRefreshComplete_t;
```

Its main parameters are as follows:

Connection_Handle	Connection index.
-------------------	-------------------

MSG_BLE_PAIR_USER_PASSKEYREQ_IND

This is a message requesting the local to input the PIN code during pairing. For the paring method that requires the local to input the PIN code, input the PIN code to complete the paring when this message is received. This message has only one connection index parameter, Connection_Handle.

MSG_BLE_PAIR_USER_PASSKEY_DISPLAY_IND

This is a message requesting the local to input the PIN code during pairing. For the paring method that requires the local to input the PIN code, the key is specified by the peer and the local only needs to display it when this message is received. Its data is a structure, as described below.

```
typedef struct __attribute__((packed))
{
    uint16_t conn_handle;
    uint8_t passkey[4];
} st_passkey_display_event;
```

Its main parameters are as follows:

Connection_Handle	Connection index.
passkey[4]	Displayed PIN code value, ranging from 0 to 999999, in little-endian order.

MSG_BLE_PAIR_USER_PASSKEYREQ_CONF_IND

This is a message that displays a confirmation code and requires the local device for confirmation during secure pairing. In the secure pairing mode, both devices will generate a confirmation code and display it. Users need to compare and confirm. This message is used to display the confirmation code and require the local device for comparison and confirmation. Its data is a structure, as described below.

```
typedef struct __attribute__((packed))
{
    uint16_t conn_handle;
    uint8_t passkey[4];
} st_passkey_display_event;
```

Its main parameters are as follows:

Connection_Handle	Connection index.
passkey[4]	Confirmation code.

MSG_BLE_PAIR_COMPLETED_IND

This is a message indicating that the pairing is successful. This message is reported after the underlying protocol stack has completed the pairing procedure and obtained the binding information. Its data is a structure, as described below.

```
typedef struct __attribute__((packed))
{
    uint16_t conn_handle;
    uint8_t key_size;
    uint8_t authenticated;
    uint8_t authorized;
    uint8_t peer_addr_type;
    uint8_t peer_addr[6];
    uint8_t peer_irk[16];
    uint8_t local_irk[16];
    uint8_t ltk[16];
    uint8_t rand[8];
    uint16_t ediv;
} st_pair_complete_event;
```

Its main parameters are as follows:

Connection_Handle	Connection index.
key_size	Effective key LTK length, ranging from 7 to 16. The actual LTK length is 16. If the effective length is less than 16, complement by 0.
authenticated	Authenticated or not.
authorized	Authorized or not.
peer_addr_type	Peer device identity address type.
peer_addr[6]	Peer device identity address.
peer_irk[16]	Peer device IRK.
local_irk[16]	Local device IRK.
ltk[16]	LTK key, which is required for the next encrypted connection.
rand[8]	Random number RAND parameter, which is required for the next encrypted connection.
ediv	EDIV parameter, which is required for the next encrypted connection.

MSG_BLE_PAIR_FAIL_IND

This is a message indicating that the pairing is failed. This message mainly indicates the pairing failure and its reason. Its data is a structure, as described below.

```
typedef struct __attribute__((packed))
{
    uint16_t conn_handle;
    uint8_t reason;
} st_pair_failed_event;
```

Its main parameters are as follows:

Connection_Handle	Connection index.
reason	Pairing failure reason.

MSG_LTK_REQ_WHEN_RECONNECT_AFTER_PAIR_IND

This is a message requesting LTK, which is only available for the Slave role devices. When the encryption is enabled for the connection, the underlying protocol stack will report this message to request LTK. At this point, the LTK data needs to be sent to the protocol stack using the rom_gap_api_long_term_key_request_reply to complete the encryption. The data is a structure, as described below.

Starting encryption is initiated by the master devices by sending the LL_ENC_REQ control packet, which contains the random number RAND parameter and the EDIV parameter in the following structure. These two parameters are also the parameters with the same name in the binding message when the pairing is completed. For the traditional pairing mode, these two parameters can be used to index the LTK, while for secure pairing mode, these two parameters are all zeros and can only be indexed to the corresponding LTK by the device address.

```
typedef struct __attribute__((packed))
{
    uint16_t conn_handle;
    uint8_t rand[8];
    uint16_t ediv;
} st_ltk_req_event;
```

Its main parameters are as follows:

Connection_Handle	Connection index.
rand[8]	Random number RAND parameter.
ediv	EDIV parameter.

MSG_BLE_ADV_REPORT_IND

This is a message reporting advertising data. After the device starts scanning, this message reports that the advertising data has been scanned. Multiple advertising data may be reported at a time, so the data format is special. The first two parameters are fixed, and the last six parameters may have multiple sets (the number of sets is the number of advertising data, Num_Reports). The specific parameter structure is described below.

Subevent_Code	HCI event message code.
Num_Reports	The data after this parameter may be multiple advertising data. This parameter is the number of advertising data.
Event_Type[i]	Advertising type of the i-th advertising, 5 types in total: HCI_ADVERTISING_REPORT_EVENT_TYPE_ADV_IND = 0x00, HCI_ADVERTISING_REPORT_EVENT_TYPE_ADV_DIRECT_IND = 0x01, HCI_ADVERTISING_REPORT_EVENT_TYPE_ADV_SCAN_IND = 0x02, HCI_ADVERTISING_REPORT_EVENT_TYPE_ADV_NONCONN_IND = 0x03, HCI_ADVERTISING_REPORT_EVENT_TYPE_SCAN_RSP = 0x04.

Address_Type[i]	Advertising device address type of the i-th advertising.
Address[i]	Advertising device address of the i-th advertising, with a fixed length of 6 bytes.
Length[i]	Advertising data length of the i-th advertising.
Data[i]	Advertising data of the i-th advertising, the length of which is the previous parameter Length[i].
RSSI[i]	Signal strength of the i-th advertising, which is a signed 8-bit data, indicating a negative value.

MSG_BLE_HIGH_DUTY_DIRECT_ADV_END_IND

This is a message reporting the end of the high duty cycle directed advertising timeout (the timeout period is 1.28 seconds). If no connection request is received to enter the connection state within 1.28 seconds, the high duty cycle directed advertising will automatically stop. This message has no additional data.

Examples

General Interfaces

Set Device Address

The example program for setting the device address, including random address and public address, is as follows:

```
uint8 pulic_addr[6] = {0x66,0x55,0x44,0x33,0x22,0x11};
uint8 random_addr[6] = {0x16,0x15,0x14,0x13,0x12,0x11};
// Set device public address
rom_gap_api_set_public_device_address(pulic_addr);
// Set device random address
rom_gap_api_set_random_device_address(random_addr);
```

Manage White List

The simple example program for adding to, removing from and clearing white list is as follows:

```
uint8 public_addr[6] = {0x13, 0x71, 0xda, 0x7d, 0x1a, 0x00};
// Add device to white list:
rom_gap_api_add_device_to_white_list(GAP_WHITE_LIST_ADDRESS_TYPE_PUBLIC,
public_addr);
// Remove device from white list:
rom_gap_api_remove_device_from_white_list(GAP_WHITE_LIST_ADDRESS_TYPE_PUBLIC,
public_addr);
// Clear white list:
rom_gap_api_clear_white_list();
```

Advertising

Start Advertising

Users can customize the advertising type to be started and advertising data according to their applications. For details, refer to the API description in the previous section. The simple example program for setting the advertising parameters and advertising message data and enabling the advertising is as follows:

```
// Set advertising parameters:
stGapSetAdvertisingParameters_t stAdvParam =
{
    .u16AdvertisingIntervalMin625us = 200 * 8 / 5,
    .u16AdvertisingIntervalMax625us = 200 * 8 / 5,
```

```
.enumAdvertisingType = GAP_ADVERTISING_TYPE_ADV_IND,
.enumOwnAddressType = GAP_OWN_ADDRESS_TYPE_PUBLIC,
.u8AdvertisingChannelMap = 7,
.enumAdvertisingFilterPolicy = GAP_ADV_FILTER_POLICY_CONN_ALL_SCAN_ALL,
};
rom_gap_api_set_advertising_parameters(0, &stAdvParam);
// Set the advertising data and scan response data:
uint8_t pu8AdvData[8] =
{
    2, GAP_ADTYPE_FLAGS,
    GAP_ADTYPE_FLAGS_GENERAL | GAP_ADTYPE_FLAGS_BREDR_NOT_SUPPORTED,
    4, GAP_ADTYPE_LOCAL_NAME_SHORT, 'R','C','U'
};
rom_gap_api_set_advertising_data(0, pu8AdvData, sizeof(pu8AdvData));
uint8_t pu8ScanResponseData[19] =
{
    2, // Length of this data
    GAP_ADTYPE_FLAGS,
    GAP_ADTYPE_FLAGS_GENERAL | GAP_ADTYPE_FLAGS_BREDR_NOT_SUPPORTED,
    9, GAP_ADTYPE_LOCAL_NAME_COMPLETE, 'R','C','U',' ','D','E','M','O'
};
rom_gap_api_set_scan_response_data(0, pu8ScanResponseData,
sizeof(pu8ScanResponseData));
// Enable advertising:
rom_gap_api_set_advertising_enable(0, true);
// Disable advertising:
rom_gap_api_set_advertising_enable(0, false);
```

Enable White List Filtering during Advertising

To enable the white list filtering during advertising, it must first set a white list and add the device addresses to be filtered to the white list. For details, refer to the `rom_gap_api_add_device_to_white_list`. Then set the `enumAdvertisingFilterPolicy` parameter to one of the following values according to the needs of filter scan or connection request when setting the advertising parameters. For example, if a device only desires to be connected and scanned by devices in the white list, set this parameter to the third value.

```
GAP_ADV_FILTER_POLICY_CONN_ALL_SCAN_WHITELIST = 0x01,
GAP_ADV_FILTER_POLICY_CONN_WHITELIST_SCAN_ALL= 0x02,
GAP_ADV_FILTER_POLICY_CONN_WHITELIST_SCAN_WHITELIST =0x03,
```

Enable Privacy during Advertising

Enabling the privacy function includes two aspects:

1. Use resolvable private address RPA for advertising;
2. Enable the private address resolution, which can respond to a scan or connection request using RPA.

For these two functions, it is required to first set a resolving list and use the relevant APIs to add the peer device identity address and IRK and the local IRK to the resolving list, which are necessary for generating the local RPA or resolving the peer RPA.

For implementing function 1, advertising using RPA, it is required to set the advertising parameter `enumOwnAddressType` to the third or fourth value as follows, and set the parameters `enumPeerAddressType` and `pu8PeerAddress[6]` according to the peer device identity address, and then start the advertising. The underlying protocol stack uses the local IRK to generate RPA as the advertising address.

```
GAP_OWN_ADDRESS_TYPE_PUBLIC = 0x00, GAP_OWN_ADDRESS_TYPE_RANDOM = 0x01, GAP_
OWN_ADDRESS_TYPE_RESOLVABLE_OR_PUBLIC = 0x02, GAP_OWN_ADDRESS_TYPE_RESOLVABLE_
OR_RANDOM = 0x03,
```

For implementing function 2, which can respond to a scan or connection request using RPA, it is required to call the relevant API to enable the private address resolution before the advertising starts. Then, after starting the advertising, if a scan or connection request using RPA is received, the RPA will be resolved to obtain its identity address before deciding whether to respond. The following is an example code.

```
// Add device to resolving list:
uint8_t peer_addr_type = GAP_INITIATOR_PEER_ADDRESS_TYPE_PUBLIC;
uint8_t peer_addr[6] = {0xcd, 0x0e, 0x5f, 0x6c, 0x4a, 0x04};
uint8_t peer_irk[16] = {0x40, 0x69, 0x48, 0x36, 0xc6, 0x4b, 0xe0, 0xd1, 0x1c,
0x9a, 0xfe, 0x7d, 0x3e, 0xa2, 0xc7, 0xf4};
uint8_t local_irk[16] = {0x5a, 0x45, 0xe7, 0xa4, 0x57, 0x1d, 0x7f, 0x36, 0x61,
0x30, 0x7e, 0xfa, 0xce, 0xfc, 0xe2, 0x58};
rom_gap_api_add_device_to_resolving_list(peer_addr_type, peer_addr, peer_irk,
local_irk);
// Device advertising address uses RPA:
stGapSetAdvertisingParameters_t stAdvParam =
{
    .u16AdvertisingIntervalMin625us = 200 * 8 / 5,
    .u16AdvertisingIntervalMax625us = 200 * 8 / 5,
    .enumAdvertisingType = GAP_ADVERTISING_TYPE_ADV_IND,
    .enumOwnAddressType = GAP_OWN_ADDRESS_TYPE_RESOLVABLE_OR_PUBLIC,
    .enumPeerAddressType = peer_addr_type,
    .u8AdvertisingChannelMap = 7,
    .enumAdvertisingFilterPolicy = GAP_ADV_FILTER_POLICY_CONN_ALL_SCAN_ALL,
};
memcpy(stAdvParam.enumPeerAddress, peer_addr, 6);
rom_gap_api_set_advertising_parameters(0, &stAdvParam);
// Enable private address resolution:
rom_gap_api_set_addr_resolution_enable (1);
```

Scanning

Start Device Scanning

Users can customize the scanning parameters according to their applications. In principle, the larger the window, the faster the required information can be scanned, and the greater the corresponding power consumption, the maximum can be equal to interval. In this case, scanning is performed all the time. For specific parameters, refer to the previous API description. Therefore if the corresponding parameters have been configured, the MCU can start the device for scanning. The example code for setting scanning parameters and enable scanning is as follows:

```
// Set scan parameters:
stGapSetScanParameters_t stParam =
{
    .enumScanType = GAP_SCAN_TYPE_ACTIVE,
    .u8ScanChannelMap = 7,
    .u16ScanInterval625us = 200 * 8 / 5, // 200ms
    .u16ScanWindow625us = 100 * 8 / 5, // 100ms
    .enumOwnAddressType = GAP_OWN_ADDRESS_TYPE_PUBLIC,
    .enumScanningFilterPolicy = GAP_SCAN_FILTER_POLICY_ACCEPT_ALL,
};
rom_gap_api_set_scan_parameters(&stParam);
// Enable scanning:
rom_gap_api_set_scan_enable(true, true);
```

```
// Disable scanning:
rom_gap_api_set_scan_enable(false, true);
```

Enable White List Filtering during Scanning

To enable the white list filtering during scanning, it must first set a white list and add the device addresses to be filtered to the white list. Then set the filtering rule parameter to use white list filtering as shown below when setting the scan parameters. In this way, when the scanning is started again, the scan requests are sent only to the devices in the white list.

```
stParam.enumScanningFilterPolicy=GAP_SCAN_FILTER_POLICY_ACCEPT_WHITELIST;
```

Enable Privacy during Scanning

Enabling the privacy function includes two aspects:

1. Use resolvable private address RPA when sending scan requests
2. Enable the private address resolution. When receiving an advertising with AdvA using RPA, the Link layer resolves the RPA using the resolving list, and then determines whether to send the scan request according to the white list filtering rule (the objects to be filtered in the white list are the resolved identity addresses).

For these two functions, it is also required to first set a resolving list and use the relevant APIs to add the peer device identity address and IRK and the local IRK to the resolving list, which are necessary for generating the local RPA or resolving the peer RPA. In addition, for function 1, it is required to set its own address to the private address type before starting the scanning; For function 2, it is required to enable the private address resolution before starting the scanning. The following is an example code:

```
// Add device to resolving list:
uint8_t peer_addr_type = GAP_INITIATOR_PEER_ADDRESS_TYPE_PUBLIC;
uint8_t peer_addr[6] = {0xcd, 0x0e, 0x5f, 0x6c, 0x4a, 0x04};
uint8_t peer_irk[16] = {0x40, 0x69, 0x48, 0x36, 0xc6, 0x4b, 0xe0, 0xd1, 0x1c,
0x9a, 0xfe, 0x7d, 0x3e, 0xa2, 0xc7, 0xf4};
uint8_t local_irk[16] = {0x5a, 0x45, 0xe7, 0xa4, 0x57, 0x1d, 0x7f, 0x36, 0x61,
0x30, 0x7e, 0xfa, 0xce, 0xfc, 0xe2, 0x58};
rom_gap_api_add_device_to_resolving_list (peer_addr_type, peer_addr, peer_irk,
local_irk);
// Device scan address uses RPA:
stGapSetScanParameters_t stParam =
{
    .enumScanType = GAP_SCAN_TYPE_ACTIVE,
    .u8ScanChannelMap = 7,
    .u16ScanInterval625us = 200 * 8 / 5, // 200 ms
    .u16ScanWindow625us = 100 * 8 / 5, // 100 ms
    .enumOwnAddressType = GAP_OWN_ADDRESS_TYPE_RESOLVABLE_OR_PUBLIC,
    .enumScanningFilterPolicy = GAP_SCAN_FILTER_POLICY_ACCEPT_ALL,
};
rom_gap_api_set_scan_parameters(&stParam);
// Enable private address resolution:
rom_gap_api_set_addr_resolution_enable (1);
```

Obtain Scanned Privacy Data

After the scan is started, if an advertising is detected, the advertising data will be reported through the message MSG_BLE_ADV_REPORT_IND. The advertising message format is described above. The following is an example code for resolving the message, pu8Buf is the advertising data buffer:

```
case MSG_BLE_ADV_REPORT_IND:
{
```

```
uint8_t i;
uint8_t u8ReportsNum = pu8Buf[1];
uint8_t u8NextReportOffset = 2;
uint8_t u8DataLength;
uint16_t u16ReportSize;
for (i = 0; i < u8ReportsNum; i++)
{
    // ADV_Type(1) + Addr_Type(1) + Addr(6) + DataLen(1) + Data(variable) +
    // RSSI(1)
    u16ReportSize = 1 + 1 + 6 + 1 + u8DataLength + 1;
    // Advertising type
    uint8_t enumAdvType = pu8Buf[u8NextReportOffset];
    // Advertising address type
    uint8_t u8AddrType = pu8Buf[u8NextReportOffset + 1];
    // Advertising address
    uint8_t* pu8Addr = &pu8Buf[u8NextReportOffset + 2];
    // Advertising data length
    u8DataLength = pu8Buf[u8NextReportOffset + 8];
    // Advertising data
    uint8_t* pu8Data = &pu8Buf[u8NextReportOffset + 9];
    // Advertising signal strength
    int8_t s8Rssi = pu8Buf[u8NextReportOffset + 9 + u8DataLength];
    PRINTF("ADV_REPORT[index:%d]-ADV_type:%u addr_type:%u rssi: %d", i,
           enumAdvType, u8AddrType, s8Rssi);
    print_hex("ADDR: ", pu8Addr, 6);
    print_hex("DATA: ", pu8Data, u8DataLength);
    u8NextReportOffset += u16ReportSize;
}
break;
}
```

Peripheral

Enter Connection

After the device starts advertising, it is connected by other devices and enters the connection state. In this case, the device plays the peripheral slave role. At this point, the message callback function will report the message MSG_BLE_CONNECTED_IND. The following is an example code for reporting MSG_BLE_CONNECTED_IND in the message callback function:

```
case MSG_BLE_CONNECTED_IND:
{
    stHciEventParamVendorConnectionComplete_t* pstEvent =
    (stHciEventParamVendorConnectionComplete_t*)pu8Buf;
    // Connection index and role
    PRINTF("connHandle:0x%x,role=%d", pstEvent->Connection_Handle, pstEvent->
           Role);
    // Peer device address
    Printf_hex("Peer_Address", pstEvent->Peer_Address, 6);
    // pstEvent->Peer_Address is RPA, the following is the resolved identity
    // address
    if (pstEvent->Peer_RPA_Resolved)
    {
        // Peer identity address
        Printf_hex("Peer_Address", pstEvent->Peer_Identify_Address, 6);
    }
    // Connection parameters
    PRINTF("Interval:%d,Latency:%d Timeout:%d", pstEvent->Conn_Interval,
```

```

        pstEvent->Conn_Latency, pstEvent->Supervision_Timeout);
    // Local address type and whether it is RPA
    PRINTF("Own_Address_Type:%d,Be_RPA=%d", pstEvent->Own_Address_Type,
        pstEvent->Own_Address_Be_RPA);
    break;
}

```

Update Connection Parameters

After the connection is established, the slave device can also call the corresponding API to request the connection parameter update. The following is an example code:

```

stGapConnectionUpdate_t stParam =
{
    .ul6ConnIntervalMin1250us = 10,
    .ul6ConnIntervalMax1250us = 20,
    .ul6ConnLatency = 0,
    .ul6SupervisionTimeout10ms = 100,
};
rom_gap_api_connection_parameters_update (ul6ConnHandle, &stParam);

```

Only after the master accepts the connection parameter update request sent by the slave at the LLCAP layer, can the master initiate the actual connection parameter update procedure. The following message is the result of the master response, with result 0 indicating acceptance:

```

case MSG_BLE_LLCAP_CONN_UPDATE_RSP_IND:
{
    st_llcap_conn_update_rsp_event* pstEvt = (st_llcap_conn_update_rsp_event*)
                                              pu8Buf;
    PRINTF("Peer device accept or reject new connection parameter, result = %d",
        pstEvt->result);
    break;
}

```

If the master accepts the connection parameter update request from the slave, the master initiates the actual connection parameter update procedure. After the update is completed, the following message is reported, indicating the updated connection parameters.

```

case MSG_BLE_CONNECTION_UPDATE_COMPLETE_IND:
{
    stHciEventParamLEConnectionUpdateComplete_t* pstEvt
    =(stHciEventParamLEConnectionUpdateComplete_t*) pu8Buf;
    PRINTF ("[MSG_BLE_CONNECTION_UPDATE_COMPLETE_IND] Conn_Interval = 0x%X,
        Conn_Latency = 0x%X, Supervision_Timeou = 0x%X\n", pstEvt->Conn_
        Interval, pstEvt->Conn_Latency, pstEvt->Supervision_Timeout);
    break;
}

```

Disconnect

After the connection is established, the slave device can also call the corresponding API to actively disconnect the connection. The following is an example code:

```

rom_gap_api_disconnect(ul6ConnHandle, 0x13);

```

Whether the local device or the peer device is disconnected, or the connection is disconnected due to other reasons, the following message is reported, indicating that the connection is disconnected and the reason for the disconnection.

```

case MSG_BLE_DISCONNECTED_IND:
{
    stHciEventParamDisconnectionComplete_t* pstEvt =
    (stHciEventParamDisconnectionComplete_t*) pu8Buf;

```

```
    PRINTF ("[MSG_BLE_DISCONNECTED_IND] Handle = 0x%X, reason = 0x%X\n", pstEvt->
        Connection_Handle, pstEvt->Reason);
    break;
}
```

Pairing and Encryption

The MCU supports two pairing modes, traditional pairing mode and secure pairing mode. It also supports several pairing methods, including OOB, Just Work, password input, numerical comparison (only available for secure pairing mode). The pairing mode and pairing method are determined by the pairing parameters of the two devices. For details, refer to the Appendix.

Set Pairing Parameters

1. Set the OOB traditional pairing method

The callback function for obtaining OOB data should be registered first. During pairing, the protocol stack queries the OOB data using this callback function. When the OOB data is obtained, the OOB flag is set. If the OOB flag is also set for the peer during the exchange of pairing information, the OOB method is selected for pairing, but information such as the device I/O capability is ignored.

```
// Implement callback function:
int get_oob_data_callback (uint8 address_type, uint8 addr[6], uint8 oob_
data[16])
{
    if(address match)
    {
        // Determined by the address, if the address matches, set OOB data and
        // return 1.
        for(i=0; i<16; i++)
        {
            oob_data[i]=xx; // Set OOB data
        }
        return 1;
    }
    else
    {
        // If the address does not match, return 0.
        return 0;
    }
}

// Register callback function:
rom_gap_api_register_oob_data_callback(get_oob_data_callback);
```

2. Set the password pairing method

If OOB is not used (the OOB callback function is not registered), the pairing method is determined according to the I/O capability of the two devices. If Keyboard Only is used on one side, and Display Only is used on the other side, the password pairing method is selected. The password is displayed on the Display Only side, and the password is entered on the Keyboard Only side. This method is the same for both traditional and secure pairing modes.

Set the parameters for the Display Only side:

```
// Traditional pairing mode:
rom_gap_api_update_pair_para(IO_CAPABILITY_DISPLAY_ONLY, SM_AUTHREQ_BONDING |
SM_AUTHREQ_MITM_PROTECTION, 16);
// Secure pairing mode:
rom_gap_api_update_pair_para(IO_CAPABILITY_DISPLAY_ONLY, SM_AUTHREQ_BONDING |
SM_AUTHREQ_MITM_PROTECTION | SM_AUTHREQ_SECURE_CONNECTION, 16);
```


During the pairing process, the Display Only device displays the password message “MSG_BLE_PAIR_USER_PASSKEY_DISPLAY_IND:” with the 6-bit decimal PIN data that needs to be entered by the peer device:

```
case MSG_BLE_PAIR_USER_PASSKEY_DISPLAY_IND:
{
    st_passkey_display_event* pstEvt = (st_passkey_display_event*) pu8Buf;
    PRINTF ("MSG_BLE_PAIR_USER_PASSKEY_DISPLAY_IND passkey = %d, Handle = 0x%X",
        rom_little_endian_read_32 (pstEvt->passkey, 0), pstEvt->conn_handle);
    break;
}
```

Set the parameters for the Keyboard Only side

```
// Traditional pairing mode:
rom_gap_api_update_pair_para(IO_CAPABILITY_KEYBOARD_ONLY, SM_AUTHREQ_BONDING |
    SM_AUTHREQ_MITM_PROTECTION, 16);
// Secure paring mode:
rom_gap_api_update_pair_para(IO_CAPABILITY_KEYBOARD_ONLY, SM_AUTHREQ_BONDING |
    SM_AUTHREQ_MITM_PROTECTION | SM_AUTHREQ_SECURE_CONNECTION, 16);
```

During the pairing process, the Keyboard Only device displays a message requesting a password, MSG_BLE_PAIR_USER_PASSKEYREQ_IND, as follows:

```
case MSG_BLE_PAIR_USER_PASSKEYREQ_IND:
{
    uint16_t u16ConnectionHandle = * ((uint16_t*) pu8Buf);
    PRINTF ("MSG_BLE_PAIR_USER_PASSKEYREQ_IND Handle = 0x%X", u16ConnectionHandle);
    break;
}
```

After receiving this message, the device needs to enter the 6-bit decimal PIN code value u32Passkey displayed by the peer device and send it to the protocol stack using the relevant API to complete the pairing:

```
rom_gap_api_input_passkey_during_pair(u16ConnHandle, u32Passkey);
```

3. Set the numerical comparison pairing method

In the secure pairing mode, if the I/O capability of both devices is Display YesNo, the numerical comparison pair method is used. During the pairing process, both devices display a value, which needs to be compared and confirmed by the user to complete the pairing.

Set the pairing parameters:

```
rom_gap_api_update_pair_para(IO_CAPABILITY_DISPLAY_YES_NO, SM_AUTHREQ_BONDING
    | SM_AUTHREQ_MITM_PROTECTION | SM_AUTHREQ_SECURE_CONNECTION, 16)
```

During the pairing process, the two devices display the MSG_BLE_PAIR_USER_PASSKEYREQ_CONF_IND message containing the 6-bit decimal value to be confirmed by the peer device:

```
case MSG_BLE_PAIR_USER_PASSKEYREQ_CONF_IND:
{
    st_passkey_display_event* pstEvt = (st_passkey_display_event*) pu8Buf;
    PRINTF ("MSG_BLE_PAIR_USER_PASSKEYREQ_CONF_IND Number= %d, Handle = 0x%X",
        rom_little_endian_read_32 (pstEvt->passkey, 0), pstEvt->conn_handle);
    break;
}
```

The device displays the value and the user compares it with the value of the peer device to see if the value is consistent. The comparison result is sent to the protocol stack using the relevant API to complete the pairing:

```
rom_gap_api_sm_numeric_comparison_confirm(u16ConnHandle, confirm_sucess);
```

4. Set the Just Work pairing method

If the I/O capability of one side is No Input No Output, the Just Work pairing method is used (the same for traditional pairing and secure pairing modes).

Set the parameters for the Just Work pairing method:

```
// Traditional pairing mode:
rom_gap_api_update_pair_para(IO_CAPABILITY_NO_INPUT_NO_OUTPUT, SM_AUTHREQ_
BONDING, 16);
// Secure pairing mode:
rom_gap_api_update_pair_para(IO_CAPABILITY_NO_INPUT_NO_OUTPUT, SM_AUTHREQ_
BONDING | SM_AUTHREQ_SECURE_CONNECTION, 16);
```

Start Pairing

As a slave device, it cannot directly initiate pairing, but can only call the related APIs to send security requests to the master device to enable the master device to start pairing:

```
rom_gap_api_send_security_req (u16ConnHandle);
```

After the pairing is completed, the pairing completion message, MSG_BLE_PAIR_COMPLETED_IND, is sent, which carries the binding information and needs to be saved for the next encryption or setting the private address resolving list:

```
case MSG_BLE_PAIR_COMPLETED_IND:
{
    st_pair_complete_event* pstPairEvt = (st_pair_complete_event*) pu8Buf;
    PRINTF ("connHdl = 0x%X,peer_addr_type =%d", pstPairEvt->conn_handle,
        pstPairEvt->peer_addr_type);
    PRINTF_hex ("peer_address", pstPairEvt->peer_addr,6);
    PRINTF_hex ("peer_IRK", pstPairEvt->peer_irk,16);
    PRINTF_hex ("local_IRK", pstPairEvt->local_irk,16);
    PRINTF_hex ("LTK", pstPairEvt->ltk,16);
    PRINTF_hex ("RAND", pstPairEvt->rand,8);
    PRINTF ("EDIV = 0x%x",pstPairEvt->ediv);
    PRINTF ("Keysize = %d, authenticated=%d",pstPairEvt->key_size,pstPairEvt->
        authenticated);
    break;
}
```

Start Encryption

If the device has been paired with the peer device, the previously saved binding information should first be loaded after the next reconnection, as follows:

```
stGapBondInfo_t stBondinfo =
{
    .u16Ediv= 0x58E6,
    .pu8Rand = {0x13, 0x02, 0xF1, 0xE0, 0xDF, 0xCE, 0xBD, 0xAC},
    .pu8Ltk = {0xbf, 0x01, 0xfb, 0x9d, 0x4e, 0xf3, 0xbc, 0x36, 0xd8, 0x74, 0xf5,
        0x39, 0x41, 0x38, 0x68, 0x4c},
    .u8KeySize=16,
    .u8Authenticated=1,
    .u8Authorized = 0,
};
rom_gap_api_set_bond_info_for_setup_encryption (connHdl, &stBondinfo);
```

Then, the relevant API is called to send the security request to the master device, and the master device will start encryption:

```
rom_gap_api_send_security_req (u16ConnHandle)
```

During the process of starting encryption, the message for requesting LTK key, MSG_LTK_REQ_WHEN_RECONNECT_AFTER_PAIR_IND, will be sent. At this point, the relevant API needs to be called to transfer the key pu8LTK saved when the pairing is completed to the protocol stack to complete the encryption process, as follows:

```
case MSG_LTK_REQ_WHEN_RECONNECT_AFTER_PAIR_IND:
{
    st_ltk_req_event* pstEvt = (st_ltk_req_event*) pu8Buf;
    PRINTF_hex ("RAND", pstEvt->rand,8);
    PRINTF ("EDIV = 0x%x",pstEvt->ediv);
    // Set LTK key to protocol stack
    rom_gap_api_long_term_key_request_reply(pstEvt->conn_handle, pu8LTK);
    break;
}
```

After the encryption state is entered, the message MSG_BLE_ENCRYPTED_CHANGED_IND is reported, where the third parameter indicates the encryption state, as follows:

```
case MSG_BLE_ENCRYPTED_CHANGED_IND:
{
    stHciEventParamEncryptionChange_t* pstEvt = (stHciEventParamEncryptionChange_
                                                t*) pu8Buf;
    PRINTF ("Handle = 0x%X,Status = 0x%X Encryption_Enabled = 0x%X\n", pstEvt->
        Connection_Handle, pstEvt->Status, pstEvt->Encryption_Enabled);
    break;
}
```

Central

Establish Connection

The device that enters the connection from the initiating state is the master device. The following is an example code for entering the initiating state to create a connection:

```
stGapCreateConnection_t stParam =
{
    .u8ScanChannelMap = 7,
    .u16ScanInterval625us = 200 * 8 / 5,
    .u16ScanWindow625us = 100 * 8 / 5,
    .enumInitiatorFilterPolicy = GAP_INITIATOR_FILTER_POLICY_WHITELIST_IS_NOT_
        USED,
    .enumPeerAddressType = GAP_INITIATOR_PEER_ADDRESS_TYPE_PUBLIC,
    .pu8PeerAddress = {0xcd, 0x0e, 0x5f, 0x6c, 0x4a, 0x04},
    .enumOwnAddressType = GAP_OWN_ADDRESS_TYPE_PUBLIC,
    .u16ConnIntervalMin1250us = 50 * 4 / 5,
    // 50ms
    .u16ConnIntervalMax1250us = 50 * 4 / 5,
    // 50ms
    .u16ConnLatency = 0,
    .u16SupervisionTimeout10ms = 200, // 2s
};
rom_gap_api_create_connection(&stParam);
```

To enable the white list filtering when creating a connection, it must first set a white list and add the device addresses to be filtered to the white list. Then set the filtering rule to use white list filtering when setting the initiating state parameters, as follows:

```
stParam.enumInitiatorFilterPolicy = GAP_INITIATOR_FILTER_POLICY_WHITELIST_IS_
    USED;
```

Similar to advertising or scanning, enabling privacy when creating a connection also includes two aspects:

1. Use resolvable private address RPA, and use RPA when sending connection requests.
2. Enable the private address resolution. When receiving an advertising with AdvA using RPA, the RPA should first be resolved and then determines whether to send the connection request according to the white list filtering rule (the objects to be filtered in the white list are the resolved identity addresses).

For these two functions, it is also required to first set a resolving list and use the relevant API to add the identity address of the peer device and IRK and local RK to the resolving list, which are necessary for generating the local RPA or resolving the peer RPA.

For function 1, it is required to set its own address to the private address type before starting connection creation.

For function 2, it is required to enable private address resolution before starting connection creation.

The following is an example code. Add devices to the resolving list:

```
uint8_t peer_addr_type = GAP_INITIATOR_PEER_ADDRESS_TYPE_PUBLIC;
uint8_t peer_addr[6] = {0xcd, 0x0e, 0x5f, 0x6c, 0x4a, 0x04};
uint8_t peer_irk[16] = {0x40, 0x69, 0x48, 0x36, 0xc6, 0x4b, 0xe0, 0xd1, 0x1c,
0x9a, 0xfe, 0x7d, 0x3e, 0xa2, 0xc7, 0xf4};
uint8_t local_irk[16] = {0x5a, 0x45, 0xe7, 0xa4, 0x57, 0x1d, 0x7f, 0x36, 0x61,
0x30, 0x7e, 0xfa, 0xce, 0xfc, 0xe2, 0x58};
rom_gap_api_add_device_to_resolving_list(peer_addr_type, peer_addr, peer_irk,
local_irk);
// Device scan address uses RPA:
stParam.enumOwnAddressType = GAP_OWN_ADDRESS_TYPE_RESOLVABLE_OR_PUBLIC ;
// Enable private address resolution:
rom_gap_api_set_addr_resolution_enable (1);
```

After the connection is successfully established, the message callback function will report the message MSG_BLE_CONNECTED_IND, the message content is the same as the slave mode.

Update Connection Parameters

The APIs are the same as that called by the slave to update connection parameters. The difference is that after the call, the master directly initiates the connection parameter update procedure at the Link Layer, during which no message MSG_BLE_LLCP_CONN_UPDATE_RSP_IND will be reported. After the update is completed, the update completion message MSG_BLE_CONNECTION_UPDATE_COMPLETE_IND will also be reported, which carries new connection parameters.

Disconnect

The called APIs and the reported messages are the same as those of the slave device.

Start Pairing and Encryption

For the master, the processes of setting pairing parameters and starting pairing is the same as that of the slave, and the process of starting encryption after reconnection is basically the same, except that the message requesting LTK key will not be reported during the process.

11 Bluetooth Low Energy GATT

Introduction

Bluetooth® Low Energy connections are based on the Generic Attribute Profile (GATT) protocol, which is a generic specification for sending and receiving short data segments over Bluetooth connections. This chapter introduces Bluetooth Low Energy GATT related APIs and their examples, including server and client implementations.

Bluetooth Low Energy GATT Server API Interfaces

API Interfaces

rom_gatts_api_init

Function Prototype

```
void rom_gatts_api_init (uint8_t *pu8ProfileBuf, uint16_t u16ProfileBufLen, gatts_serv_info_st  
*pstGattsServInfoBuffer, uint16_t u16GattsServInfoBufferNum);
```

Functional Description

Initialize the protocol stack on the Server.

Parameter

Parameter	Description
pu8ProfileBuf	Specify a memory area to store the profile information to be added. Generally, each device has multiple profiles. The length of this area depends on the profile to be added.
u16ProfileBufLen	Length of pu8ProfileBuf memory area.

Return Value

None.

rom_gatts_add_service_start

Function Prototype

```
uint16_t rom_gatts_api_add_service_start(bool is_uuid128, uint16_t uuid16, uint8_t* uuid128, gatt_  
serviceCBs_t *pServiceCBs);
```

Functional Description

Add a service.

Parameter

Parameter	Description
is_uuid128	Indicate whether the service UUID is 128-bit. If yes, the second parameter is not required to configure, only the third parameter is required. Otherwise, only the second parameter is required to configure.
uuid16	16-bit service UUID, configure this parameter if the service UUID is not 128-bit.
uuid128	128-bit service UUID, configure this parameter if the service UUID is 128-bit.

Parameter	Description
pServiceCBs	Callback function for characteristic read or write, which is called when characteristic read or write is executed.

Return Value

handle or 0	If successful, return the handle indicating the start of the service. Otherwise, return 0.
-------------	--

Note

This API must be used in pairs with rom_gatts_add_service_end. First call this API to add a service and then add the characteristics and descriptors contained in the service. Finally, call rom_gatts_add_service_end to complete the addition of the service.

rom_gatts_add_service_end

Function Prototype

```
uint16_t rom_gatts_api_add_service_end(void);
```

Functional Description

End the Add Service, indicating that a service has been added.

Parameter

None.

Return Value

handle or 0	If successful, return the handle indicating the end of the service. Otherwise, return 0.
-------------	--

Note

This API must be used in pairs with rom_gatts_add_service_start. First call rom_gatts_add_service_start to add a service and then add the characteristics and descriptors contained in the service. Finally, call this API to complete the addition of the service.

rom_gatts_api_add_include_service

Function Prototype

```
uint16_t rom_gatts_api_add_include_service (uint16_t u16SrvHandle, uint16_t u16SrvHandleEnd ,  
bool is_uuid128, uint16_t u16SrvUuid16, uint8_t *pu8ServUuid128);
```

Functional Description

Add a include service declaration.

Parameter

Parameter	Description
u16SrvHandle	Start handle of the include service.
u16SrvHandleEnd	End handle of the include service, that is, the handle of the last characteristic of the service to be included.
is_uuid128	Indicate whether the include service UUID is 128-bit. If yes, u16SrvUuid16 is not required to configure, only pu8ServUuid128 is required. Otherwise, only pu8ServUuid128 is required to configure.
u16SrvUuid1	16-bit include service UUID, configure this parameter if the UUID is not 128-bit.
pu8ServUuid128	128-bit service UUID, configure this parameter if the UUID is 128-bit.

Return Value

handle or 0	If successful, return the handle indicating the include service declaration. Otherwise, return 0.
-------------	---

Note

This API must be called immediately after rom_gatts_add_service_start to declare the services included in the current service.

rom_gatts_api_add_char

Function Prototype

```
uint16_t rom_gatts_api_add_char(bool is_uuid128, uint16_t uuid16, uint8_t* uuid128, uint16_t properties, uint8_t* data, uint16_t data_len);
```

Description

Add a characteristic.

Parameter

Parameter	Description
is_uuid128	Indicate whether the characteristic UUID is 128-bit. If yes, the second parameter is not required to configure, only the third parameter is required. Otherwise, only the second parameter is required to configure.
u16Uuid16	16-bit characteristic UUID, configure this parameter if the characteristic UUID is not 128-bit.
pu8Uuid128	128-bit characteristic UUID, configure this parameter if the characteristic UUID is 128-bit.
u16Properties	Characteristic properties, refer to the "Characteristic Property Description" section.
pu8Data	Characteristic value, which can be configured here for the characteristic with fixed content. Otherwise, for the characteristic whose value may change, set to NULL and then return the current characteristic value in the callback function for reading characteristic.
u16DataLen	Characteristic value length, ignored if pu8Data is NULL.

Return Value

handle or 0	If successful, return the handle indicating the characteristic value. Otherwise, return 0.
-------------	--

Note

None.

rom_gatts_api_add_char_descrip

Function Prototype

```
uint16_t rom_gatts_api_add_char_descrip(uint16_t uuid16, uint16_t properties, uint8_t * data, uint16_t data_len);
```

Functional Description

Add a characteristic descriptor.

Parameter

Parameter	Description
U16Uuid16	16-bit characteristic descriptor.
u16Properties	Characteristic descriptor properties, refer to the “Characteristic Property Description” section.
pu8Data	Characteristic value, which can be configured here for the characteristic with fixed content. Otherwise, for the characteristic whose value may change, set to NULL and then return the current characteristic value in the callback function for reading characteristic.
u16DataLen	Characteristic value descriptor length, ignored if pu8Data is NULL.

Return Value

handle or 0	If successful, return the handle indicating the characteristic value descriptor. Otherwise, return 0.
-------------	---

Note

This API must be called after rom_gatts_add_char to describe the previous characteristic.

rom_gatts_api_add_char_descrip_client_config

Function Prototype

```
uint16_t rom_gatts_api_add_char_descrip_client_config(void);
```

Functional Description

Add a Client Config characteristic descriptor.

Parameter

None.

Return Value

handle or 0	If successful, return the handle indicating the Client Config characteristic descriptor. Otherwise, return 0.
-------------	---

Note

The Client Config characteristic descriptors are often used, so setting this API can simplify the process of adding such descriptors. The Client Config characteristic descriptor must be called after rom_gatts_add_char to indicate that this is the Client Config characteristic descriptor of the previous characteristic. This descriptor is used to control whether the previous characteristic can send notify or indication. 0x0001 indicates that notify can be sent, 0x0002 indicates that indication can be sent, and 0 indicates that both cannot be sent. The value must be configured by the Client.

rom_gatts_api_set_next_attribute_handle

Function Prototype

```
bool rom_gatts_api_set_next_attribute_handle (uint16_t u16Handle);
```

Functional Description

Set the handle for the next service or characteristic to be added.

Parameter

Parameter	Description
u16Handle	Handle value to be set.

Return Value

true or false	If successful, return true. Otherwise, return false.
---------------	--

Note

This API must be called before the API for adding a service or characteristic (including characteristic descriptor), and the Handle value should be set to a value greater than the current Handle value (the return value of the API for adding a service or characteristic). Otherwise, it will fail.

rom_gatts_api_send_notify

Function Prototype

```
uint32_t rom_gatts_api_send_notify(uint16_t conn_handle, uint16_t attribute_handle, uint8_t *value, uint16_t value_len);
```

Functional Description

Send notify data.

Parameter

Parameter	Description
u16Connhandle	Handle of the current connection.
u16AttHdl	Handle of the characteristic for notify to be sent.
pu8Value	Pointer to the notify data to be sent.
u16ValLen	Length of the notify data to be sent.

Return Value

true or false	If successful, return 0. Otherwise, return non-0.
---------------	---

Note

The value of the Client Config characteristic descriptor of the characteristic for notify to be sent must be set to 0x0001 before it can be sent. In addition, the length of the data to be sent cannot be greater than MTU-3.

rom_gatts_api_send_indicate

Function Prototype

```
uint32_t rom_gatts_api_send_indicate(uint16_t conn_handle, uint16_t attribute_handle, uint8_t *value, uint16_t value_len);
```

Functional Description

Send indication data.

Parameter

Parameter	Description
u16Connhandle	Handle of the current connection.
u16AttHdl	Handle of the characteristic for indication to be sent.
pu8Value	Pointer to the indication data to be sent.
u16ValLen	Length of the indication data to be sent.

Return Value

true or false	If successful, return 0. Otherwise, return non-0.
---------------	---

Note

The value of the Client Config characteristic descriptor of the characteristic for indication to be sent must be set to 0x0002 before it can be sent. In addition, the length of the data to be sent cannot be greater than MTU-3.

(*att_get_attribute_length_callback_t)

Function Prototype

```
uint16_t (*att_get_attribute_length_callback_t)(uint16_t conn_handle, uint16_t attribute_handle);
```

Functional Description

When adding a service, it requires to specify a callback function using p_service_CBs. There are three callback functions. The callback function here is used to return the length of the characteristic to be read.

Parameter

Parameter	Description
u16Connhandle	Handle of the current connection.
u16AttHdl	Handle of the characteristic to be read.

Return Value

uint16_t	Total length of the characteristic value to be read.
----------	--

Note

When the protocol stack reads a characteristic with an unfixed characteristic value on the Client, it calls this API to obtain the maximum length of the characteristic value, and then uses the following callback function, att_read_callback_t, to obtain the characteristic value data.

(*att_read_callback_t)

Function Prototype

```
uint16_t (*att_read_callback_t)(uint16_t conn_handle, uint16_t attribute_handle, uint16_t offset, uint8_t * buffer, uint16_t buffer_size);
```

Functional Description

When adding a service, it requires to specify a callback function using p_service_CBs. There are three callback functions. The callback function here is used to obtain the characteristic value to be read.

Parameter

Parameter	Description
u16Connhandle	Handle of the current connection.
u16AttHdl	Handle of the characteristic to be read.
u16Offset	Offset of the characteristic value to be read. 0 indicates that it is read from the beginning.
pu8Buffer	Memory address used to store the characteristic value. The characteristic value needs to be copied to this address, starting from u16Offset bytes.
U16BufSize	Maximum length of memory used to store the characteristic value, which cannot be exceeded when copying.

Return Value

uint16_t	Return the copied data length.
----------	--------------------------------

Note

When the protocol stack reads a characteristic with an unfixed characteristic value on the Client, it calls this API to obtain the characteristic value data.

(*att_write_callback_t)

Function Prototype

```
uint32_t (*att_write_callback_t)(uint16_t conn_handle, uint16_t attribute_handle, uint16_t transaction_mode, uint16_t offset, uint8_t *buffer, uint16_t buffer_size);
```

Functional Description

When adding a service, it requires to specify a callback function using p_service_CBs. There are three callback functions. The callback function here is used to rewrite the characteristic value.

Parameter

Parameter	Description
u16Connhandle	Handle of the current connection.
u16AttHdl	Handle of characteristics to be rewritten.
u16TransMode	Rewrite operation mode.
u16Offset	Offset of the characteristic value to be rewritten. 0 indicates that it is rewritten from the beginning. This parameter is valid only in the Prepare write mode.
pu8Buffer	Characteristic value data to be rewritten.
u16BufSize	Data length to be rewritten.

Return Value

uint32_t	If successful, return 0. Otherwise, return non-0.
----------	---

Note

When the protocol stack rewrites a characteristic value on the Client, it calls this API to rewrite the characteristic value. In addition, when a Confirm is received from the other party after an indication is sent, this callback function will also be called to determine by u16TransMode.

Characteristic Property Description

There are several common characteristic properties, each of which is identified by a bit. A characteristic can have multiple properties. For example, a readable and writable characteristic has properties of ATT_PROPERTY_READ | ATT_PROPERTY_WRITE | ATT_PROPERTY_DYNAMIC. The complete characteristic properties are listed below.

Property Definition	Description
ATT_PROPERTY_READ(0x02)	Read property, indicating that the characteristic is readable.
ATT_PROPERTY_WRITE_WITHOUT_RESPONSE(0x04)	Write property, indicating that the characteristic is writable (unreliable write, i.e. a way that does not require the other party to determine after writing).
ATT_PROPERTY_WRITE(0x08)	Reliable write property, indicating that the characteristic is writable (reliable write, that is, a way to wait for the other party to determine after writing).

Property Definition	Description
ATT_PROPERTY_NOTIFY(0x10)	Notify property, indicating that the characteristic can send notify. (A characteristic with this property must add a Client Config characteristic descriptor).
ATT_PROPERTY_INDICATE(0x20)	Indication property, indicating that the characteristic can send indication (a characteristic with this property must add a Client Config characteristic descriptor).
ATT_PROPERTY_DYNAMIC(0x100)	Dynamic characteristic value property, indicating that the characteristic value may change dynamically. Any characteristic with this property needs to be processed in the read-write callback function. A characteristic with writable property must also set this property.
ATT_PROPERTY_AUTHENTICATION_REQUIRED(0x400)	Authentication property, indicating that the characteristic can be accessed only after pairing and encryption. The access will fail before encryption.

Transaction Mode Description in Write Characteristic Callback Function

There are five transaction modes in the callback function, as shown in the following table. The middle three modes are used for queued writes. A queued write generally involves several prepare write operations (each with a different offset position), followed by the execute write or cancel write operation.

Transaction Mode	Description
ATT_TRANSACTION_MODE_NONE(0x0)	Normal write, which immediately rewrites the characteristic values with the incoming data.
ATT_TRANSACTION_MODE_ACTIVE(0x1)	Prepare write the sent data, which needs to be stored temporarily.
ATT_TRANSACTION_MODE_EXECUTE(0x2)	Execute write, rewriting the characteristic values with all previous prepare write data.
ATT_TRANSACTION_MODE_CANCEL(0x3)	Cancel write, discarding the previous stored prepare write data.
ATT_TRANSACTION_MODE_CONFIRM_FOR_INDICAT(0x4)	After sending Indication data, a Confirm is received from the other party.

Example Code: Transparent Profile

Transparent Service Overview

The transparent profile contains only one service, a custom transparent service with a service UUID of 0xfff0. The service has the following characteristics:

Characteristic UUID	Characteristic Property	Description
0xffff2	ATT_PROPERTY_WRITE_WITHOUT_RESPONSE ATT_PROPERTY_DYNAMIC	Downlink transmission characteristic, used for data downlink. The peer sends data using the write method. It has unreliable write and dynamic characteristic value properties.
0xffff1	ATT_PROPERTY_NOTIFY ATT_PROPERTY_DYNAMIC	Uplink transmission characteristic, used for data uplink. Data is sent to the peer using the notify method. It has notify and dynamic characteristic value properties.

Characteristic UUID	Characteristic Property	Description
0x2902	ATT_PROPERTY_READ ATT_PROPERTY_WRITE ATT_PROPERTY_WRITE_WITHOUT_RESPONSE ATT_PROPERTY_DYNAMIC	Uplink transmission characteristic descriptor, used to control whether the notify function of a characteristic is enabled. It has read, reliable write, unreliable write and dynamic variable characteristic value properties.
0xff3	ATT_PROPERTY_NOTIFY ATT_PROPERTY_WRITE_WITHOUT_RESPONSE ATT_PROPERTY_DYNAMIC	Command characteristic, used to transmit custom commands, such as modify connection parameters. It has notify, unreliable write and dynamic characteristic value properties.
0x2902	ATT_PROPERTY_READ ATT_PROPERTY_WRITE ATT_PROPERTY_WRITE_WITHOUT_RESPONSE ATT_PROPERTY_DYNAMIC	Client Configuration descriptor of 0xff3 characteristic, used to control whether the notify function of a characteristic is enabled. It has read, reliable write, unreliable write and dynamic characteristic value properties.
0xff4	ATT_PROPERTY_READ ATT_PROPERTY_AUTHENTICATION_REQUIRED	Authentication property Test characteristic, used to test the characteristic with authentication property. It has read and authentication properties. This characteristic can be read only after link pairing and encryption.
0xff6	ATT_PROPERTY_READ ATT_PROPERTY_WRITE ATT_PROPERTY_DYNAMIC	Characteristic for testing Blob read and prepare write. It has read, reliable write and dynamic characteristic value properties.
0xff7	ATT_PROPERTY_INDICATE ATT_PROPERTY_DYNAMIC	Characteristic for testing indication. It has indication and dynamic characteristic value properties.
0x2902	ATT_PROPERTY_READ ATT_PROPERTY_WRITE ATT_PROPERTY_WRITE_WITHOUT_RESPONSE ATT_PROPERTY_DYNAMIC	Client Configuration descriptor of 0xff7 characteristic, used to control whether the indication function of a characteristic is enabled. It has read, reliable write, unreliable write and dynamic characteristic value properties.
0x2901	ATT_PROPERTY_READ ATT_PROPERTY_WRITE ATT_PROPERTY_DYNAMIC	User Description descriptor of 0xff7 characteristic. It has read, write and dynamic characteristic value properties.

Transparent Service Implementation Code

The detailed code for the transparent service can be found in `trx_service.c` and `trx_service.h`, where the key code is listed and explained. The UUIDs used in the transparent service are as follows:

#define TX_RX_SERVICE_UUID	0xFFF0
#define RX_CUS_UUID	0xFFF1
#define TX_CUS_UUID	0xFFF2
#define TRX_CMD_UUID	0xFFF3
#define TRX_READ_FOR_ENCPT_UUID	0xFFF4
#define TRX_BLOBREAD_PREPWRITE_TEST_UUID	0xFFF6
#define TRX_INDICATE_UUID	0xFFF7

Service Initialization

```
uint32_t trx_init_service (void)
{
    int16_t hdl = 0;
    // Here call the API to start adding the service, declare the service,
```

```
// set the service UUID and the callback function
hdl = rom_gatts_add_service_start (0, TX_RX_SERVICE_UUID, NULL, &txrxCBs);
if (hdl == 0)
{
    return RTN_FAIL;
}
// Here record the characteristic handle, so that it can be
// distinguished and found in the callback function txrx_server_hdl[RX_
// NOTIFY] = hdl;
/* Here add the characteristic Client Configuration descriptor here and record
its handle, so that it can be distinguished and found in callback function */
hdl = om_gatts_add_char_descrip_client_config();
if (hdl == 0)
{
    return RTN_FAIL;
}
txrx_server_hdl[RX_NOTIFY_CONFIG] = hdl;
/* Here add the third characteristic in the service, which is used to transmit
some custom commands and their responses. This characteristic has the
notify and write properties, and can be used for uplink and downlink data
transmission, downlink command transmission and uplink command response */
hdl = rom_gatts_add_char (0, TRX_CMD_UUID, NULL, (ATT_PROPERTY_NOTIFY | ATT_
PROPERTY_WRITE_WITHOUT_RESPONSE | ATT_PROPERTY_
DYNAMIC), NULL, 0);

if (hdl == 0)
{
    return RTN_FAIL;
}
// Record its handle as well
txrx_server_hdl[TXRX_CMD] = hdl;
// Here add the Client Configuration descriptor for the
// third characteristic and record its handle
hdl = rom_gatts_add_char_descrip_client_config();
if (hdl == 0)
{
    return RTN_FAIL;
}
txrx_server_hdl[TXRX_CMD_CONFIG] = hdl;
/* Here add the fourth characteristic in the service, which is used to
test the characteristic authentication property. Its characteristic
value has only one fixed byte, without adding the ATT_PROPERTY_DYNAMIC
property, which does not need to be processed in the callback function.
This characteristic has a read property, but can only be read after pairing
and encryption. Reading this characteristic before the connection enters
encryption will return an insufficient authentication error
(INSUFFICIENT_AUTHENTICATION). After receiving the error, the peer needs to
encrypt the connection before executing the read operation. */
hdl = rom_gatts_add_char (0, TRX_READ_FOR_ENCRYPT_UUID, NULL,
(ATT_PROPERTY_READ | ATT_PROPERTY_AUTHENTICATION_REQUIRED), &val, 1);
if (hdl == 0)
{
    return RTN_FAIL;
}
txrx_server_hdl[TXRX_READ_FOR_ENCRYPT] = hdl;
/* Here add the fifth characteristic in the service, which is used to
test Blob read and Prepare Write operations. This characteristic has
read, write and ATT_PROPERTY_DYNAMIC properties. Data reads and writes need
to be handled in the callback function. */
```

```

hdl = rom_gatts_add_char (0, TRX_BLOBREAD_PREPWRITE_TEST_UUID,
NULL,(ATT_PROPERTY_READ | ATT_PROPERTY_WRITE | ATT_PROPERTY_DYNAMIC), NULL,0);
if (hdl == 0)
{
    return RTN_FAIL;
}
// Record its handle as well
txrx_server_hdl[BLOBREAD_PREPWRITE_IDX] = hdl;
/* Here add the sixth characteristic in the service, which is used to
test Indication. This characteristic has indication and ATT_PROPERTY_
DYNAMIC properties. The confirmation of the sent indication data is handled
in the write callback function. */
hdl = gatts_add_char (0, TRX_INDICATE_UUID, NULL, (ATT_PROPERTY_INDICATE |
ATT_PROPERTY_DYNAMIC), NULL, 0);

if (hdl == 0)
{
    return RTN_FAIL;
}
txrx_server_hdl[TRX_INDICATE_IDX] = hdl;
/* Here add a user-defined descriptor for the sixth characteristic and
record its handle. This descriptor is readable and writable and needs to
be handled in the callback function. */
hdl = gatts_add_char_descrip (GATT_CHAR_USER_DESC_UUID, ATT_PROPERTY_READ |
ATT_PROPERTY_WRITE | ATT_PROPERTY_DYNAMIC,
NULL, 0);

if (hdl == 0)
{
    return RTN_FAIL;
}
txrx_server_hdl[TRX_INDICATE_USERD_IDX] = hdl;
/* The following API indicates that the transparent service configuration
is completed. This API must be called to end the configuration. Together with
the previous called gatts_add_service_start, the characteristics and
descriptors added between the two APIs belong to the current service. */
rom_gatts_add_service_end();
/* The following is to initialize the characteristic value of the fifth
characteristic and the characteristic value of the sixth characteristic
user-defined descriptor, both of which are stored in the global arrays
char_blobread_prepwrite and desp_blobread_prepwrite. */
for (i = 0; i < sizeof (char_blobread_prepwrite); i++)
{
    char_blobread_prepwrite[i] = i;
    desp_blobread_prepwrite[i] = i;
}
return 0;
}

```

Callback Functions for Transparent Service

When the transparent service is initialized, a parameter, txrxCBs, specifies the callback function, which actually specifies three callback functions.

```

gatt_serviceCBs_t txrxCBs =
{
    trx_get_attrLenCB, // get dynamic attr len
    trx_read_attrCB,   // Read callback function
    trx_write_attrCB,  // Write callback function
};

```

Where `trx_get_attrLenCB` and `trx_read_attrCB` are called in pairs and in order, that is, `trx_get_attrLenCB` is called before `trx_read_attrCB` is called.

```
/* Callback function for obtaining characteristic value length, which is
   called when the peer device reads the corresponding characteristic to
   return the length of the corresponding characteristic value. The u16AttHdl
   parameter is the handle of the characteristic to be operated. These handles
   are recorded during service initialization. Only characteristics with read
   property need to be handled using this callback function. */
uint16_t trx_get_attrLenCB (uint16_t u16ConnHdl, uint16_t u16AttHdl)
{
    /*Length of the Client Configuration descriptor for the second characteristic.
      There are three possible values for such descriptors:
      0x0000: Indicates that notify or indication of the characteristic is disabled
      0x0001: Notify is enabled
      0x0002: Indication is enabled. Therefore the length is 2*/
    if (txrx_server_hdl[RX_NOTIFY_CONFIG] == u16AttHdl)
    {
        return 2;
    }
    // Client Configuration descriptor for the third characteristic
    if (txrx_server_hdl[TXRX_CMD_CONFIG] == u16AttHdl)
    {
        return 2;
    }
    // Client Configuration descriptor for the sixth characteristic
    if (txrx_server_hdl[TRX_INDICATE_CONFIG_IDX] == u16AttHdl)
    {
        return 2;
    }
    /* Characteristic value length of the fifth characteristic.
      This characteristic value is stored in the global array
      char_blobread_prepwrite, therefore return the array length */
    if (txrx_server_hdl[BLOBREAD_PREPWRITE_IDX] == u16AttHdl)
    {
        return sizeof (char_blobread_prepwrite);
    }
    /* Characteristic value length of the user-defined descriptor for the sixth
      characteristic. This characteristic value is stored in the global array
      desp_blobread_prepwrite, therefore return the array length */
    if (txrx_server_hdl[TRX_INDICATE_USERD_IDX] == u16AttHdl)
    {
        return sizeof (desp_blobread_prepwrite);
    }
    return 0;
}

/* Callback function for obtaining characteristic value, which is called when
   the peer device reads the corresponding characteristic to obtain the
   corresponding characteristic value. The attribute_handle parameter is the
   handle of the characteristic to be operated. These handles are recorded
   during service initialization. Only characteristics with read property need
   to be handled using this callback function. */
uint16_t trx_read_attrCB (uint16_t u16ConnHdl, uint16_t attribute_handle,
uint16_t offset, uint8* buffer, uint16_t buffer_size)
{
    /* Read the contents of the Client Configuration descriptor for the second
      characteristic. The value of this descriptor is stored in the global variable
      rx_client_char_cfg */
```



```

if (txrx_server_hdl[RX_NOTIFY_CONFIG] == attribute_handle)
{
    if (buffer)
    {
        buffer[0] = rx_client_char_cfg;
        buffer[1] = 0;
    }
    return 2;
}
/* Read the contents of the Client Configuration descriptor for the third
characteristic. The value of this descriptor is stored in the global variable
txrx_cmd_char_cfg */
if (txrx_server_hdl[TXRX_CMD_CONFIG] == attribute_handle)
{
    if (buffer)
    {
        buffer[0] = txrx_cmd_char_cfg;
        buffer[1] = 0;
    }
    return 2;
}
/* Read the contents of the Client Configuration descriptor for the sixth
characteristic. The value of this descriptor is stored in the global variable
txrx_indicate_char_cfg */
if (txrx_server_hdl[TRX_INDICATE_CONFIG_IDX] == attribute_handle)
{
    if (buffer)
    {
        buffer[0] = txrx_indicate_char_cfg;
        buffer[1] = 0;
    }
    return 2;
}
/* Read the characteristic value of the fifth characteristic. The peer may
read it in Blob read mode, so the offset that may be read is not 0. This
characteristic value is stored in the global array char_blobread_prepwrite.
The buffer returns data based on the offset and length to be read by the
peer. */
if (txrx_server_hdl[BLOBREAD_PREPWRITE_IDX] == attribute_handle) //blob read
{
    int copylen = buffer_size;
    if ((offset + copylen) > sizeof (char_blobread_prepwrite))
    {
        copylen = sizeof (char_blobread_prepwrite) - offset;
    }
    if (buffer)
    {
        memcpy (buffer, char_blobread_prepwrite + offset, copylen);
    }
    return copylen;
}
/* Read the characteristic value of the user-defined descriptor for the
sixth characteristic. The peer may read it in Blob read mode, so the
offset that may be read is not 0. This characteristic value is stored
in the global array desp_blobread_prepwrite. The buffer returns
data based on the offset and length to be read by the peer. */
if (txrx_server_hdl[TRX_INDICATE_USERD_IDX] == attribute_handle) //blob read
{

```

```

    int copylen = buffer_size;
    if ((offset + copylen) > sizeof (desp_blobread_prepwrite))
    {
        copylen = sizeof (desp_blobread_prepwrite) - offset;
    }
    if (buffer)
    {
        memcpy (buffer, desp_blobread_prepwrite + offset, copylen);
    }
    return copylen;
}
return 0;
}
/*
Callback function for rewriting characteristic value, which is called when
the peer device writes the corresponding characteristic value to transmit
the corresponding characteristic value to be written. In addition, after the
Indication data is sent for the characteristic with indication property,
the protocol stack will also call this function to inform users that the
confirmation has been received when it receives a confirm packet from the other
party.
The attribute_handle parameter is the handle of the characteristic to be
operated. These handles are recorded during service initialization. Only
characteristics with read or indication property need to be handled using this
callback function.
*/
uint32_t trx_write_attrCB (uint16_t ul6ConnHdl, uint16_t attribute_handle,
uint16_t transaction_mode, uint16_t offset, uint8* buffer, uint16_t buffer_size)
{
    if (txrx_server_hdl[TX_WRITE] == attribute_handle)
    {
        /* The first characteristic receives data written by the other party,
        which is stored in a buffer with a length of buffer_size.
        txrx_server_receive is a function that processes the received data */
        txrx_server_receive (ul6ConnHdl, buffer, buffer_size);
    }
    else if (txrx_server_hdl[TXRX_CMD] == attribute_handle)
    {
        // The third characteristic receives data written by the other party.
        // txrx_cmd_receive is a function that processes the received data
        txrx_cmd_receive (ul6ConnHdl, buffer, buffer_size);
    }
    else if (txrx_server_hdl[RX_NOTIFY_CONFIG] == attribute_handle)
    {
        /* The Client config descriptor of the second characteristic receives
        the data written by the other party. The peer enables or disables
        notify for the second characteristic in this way */
        rxClientCharCfg = buffer[0];
    }
    else if (txrx_server_hdl[TXRX_CMD_CONFIG] == attribute_handle)
    {
        /* The Client config descriptor of the third characteristic receives
        the data written by the other party. The peer enables or disables
        notify for the third characteristic in this way */
        txrx_cmd_char_cfg = buffer[0];
    }
    else if (txrx_server_hdl[TRX_INDICATE_CONFIG_IDX] == attribute_handle)

```

```

{
    /* The Client config descriptor of the sixth characteristic receives
       the data written by the other party. The peer enables or disables
       Indication for the sixth characteristic in this way */
    txrx_indicate_char_cfg = buffer[0];
}
else if (txrx_server_hdl[BLOBREAD_PREPWRITE_IDX] == attribute_handle)
{
    if (ATT_TRANSACTION_MODE_NONE == transaction_mode) //normal write
    {
        /* The fifth characteristic receives the data written by the peer.
           This is a normal write mode with an offset of 0. The characteristic
           value needs to be rewritten immediately after receiving the data */
        int copylen = buffer_size;
        if (buffer_size > sizeof (char_blobread_prepwrite))
        {
            copylen = sizeof (char_blobread_prepwrite);
        }
        memcpy (char_blobread_prepwrite, buffer, copylen);
    }
    else if (ATT_TRANSACTION_MODE_ACTIVE == transaction_mode) //prepare write
    {
        /* The fifth characteristic receives the data written by the peer.
           This is a Prepare Write mode. Generally, the peer will perform multiple
           Prepare writes, each time writing the content of a different
           characteristic value field. The length and position of the field to be
           written are determined by the offset and length. After receiving such
           data, it is necessary to store it first. At this point, the corresponding
           characteristic value cannot be rewritten immediately. The peer needs
           to send an execute write before actually rewriting the characteristic
           value. */
        if (NULL == pBuffer_for_prepwrite)
        {
            pBuffer_for_prepwrite = mem_for_prepwrite;
            offset_for_prepwrite = 0;
        }
        if (NULL != pBuffer_for_prepwrite)
        {
            int copylen = buffer_size;
            if (offset >= BLE_MEM_POOL_SIZE)
            {
                copylen = 0;
            }
            else if ((offset + copylen) > BLE_MEM_POOL_SIZE)
            {
                copylen = BLE_MEM_POOL_SIZE - offset;
            }
            if (copylen && buffer)
            {
                memcpy (pBuffer_for_prepwrite + offset, buffer, copylen);
                offset_for_prepwrite += copylen;
            }
        }
    }
}
else if (ATT_TRANSACTION_MODE_EXECUTE == transaction_mode) //execute write

```

```

{
    /* The fifth characteristic receives the execute write command from the
       peer. After receiving this command, the characteristic value is
       rewritten using the previously stored peer prepare write data. */
    if (pbuffer_for_prepwrite && (offset_for_prepwrite > 0))
    {
        int copylen = offset_for_prepwrite;
        if (offset_for_prepwrite > sizeof (char_blobread_prepwrite))
        {
            copylen = sizeof (char_blobread_prepwrite);
        }
        memcpy (char_blobread_prepwrite, pbuffer_for_prepwrite, copylen);
    }
    pbuffer_for_prepwrite = NULL;
    offset_for_prepwrite = 0;
}
else if (ATT_TRANSACTION_MODE_CANCEL == transaction_mode) //cancel write
{
    /* The fifth characteristic receives the Cancel write command from the
       peer. After receiving this command, discard the previously stored
       peer prepare write data and do not perform any other operations. */
    pbuffer_for_prepwrite = NULL;
    offset_for_prepwrite = 0;
}
}
else if ((txrx_server_hdl[TRX_INDICATE_IDX] == attribute_handle) &&
(ATT_TRANSACTION_MODE_CONFIRM_FOR_INDICAT == transaction_mode))
{
    /* After the sixth characteristic sends the indication data, a confirm response
       is received from the peer. The confirmation indicates that the peer has
       processed the sent indication data. The following code is only a demo
       used to complete multiple consecutive sending of indication data to
       the peer. */
    if (g_data_indicate_ent > 0)
    {
        txrx_test_indicate_data (g_data_indicate_ent);
        g_data_indicate_ent--;
    }
}
else if (txrx_server_hdl[TRX_INDICATE_USERD_IDX] == attribute_handle)
{
    if (ATT_TRANSACTION_MODE_NONE == transaction_mode) //normal write
    {
        /* The user-defined descriptor of the sixth characteristic receives
           the data written by the peer. This is a normal write mode with an
           offset of 0. The descriptor value needs to be rewritten immediately
           after receiving the data */
        int copylen = buffer_size;
        if (buffer_size > sizeof (desp_blobread_prepwrite))
        {
            copylen = sizeof (desp_blobread_prepwrite);
        }
        memcpy (desp_blobread_prepwrite, buffer, copylen);
    }
    else if (ATT_TRANSACTION_MODE_ACTIVE == transaction_mode) //prepare write

```

```

{
    /* The user-defined descriptor of the sixth characteristic receives the
    data written by the peer. This is a Prepare Write mode. Generally,
    the peer will perform multiple Prepare writes, each time writing the
    content of a different descriptor field. The length and position of
    the field to be written are determined by the offset and length. After
    receiving such data, it is necessary to store it first. At this point,
    the corresponding descriptor cannot be rewritten immediately. The
    peer needs to send an execute write before actually rewriting the
    descriptor content. Here is the example code for temporarily storing
    the prepare write data. */
    if (NULL == pBuffer_for_prepwrite)
    {
        pBuffer_for_prepwrite = mem_for_prepwrite;
        offset_for_prepwrite = 0;
    }
    if (NULL != pBuffer_for_prepwrite)
    {
        int copylen = buffer_size;
        if (offset >= BLE_MEM_POOL_SIZE)
        {
            copylen = 0;
        }
        else if ((offset + copylen) > BLE_MEM_POOL_SIZE)
        {
            copylen = BLE_MEM_POOL_SIZE - offset;
        }
        if (copylen && buffer)
        {
            memcpy (pBuffer_for_prepwrite + offset, buffer, copylen);
            offset_for_prepwrite += copylen;
        }
    }
}
else if (ATT_TRANSACTION_MODE_EXECUTE == transaction_mode) //execute write
{
    /* The user-defined descriptor of the sixth characteristic receives the
    execute write command from the peer. After receiving this command, the
    descriptor is rewritten using the previously stored peer prepare write
    data. Here is the example code, because the descriptor is stored in
    desp_blobread_prepwrite, update the content of desp_blobread_prepwrite
    with the previously stored prepare write data */
    if (pBuffer_for_prepwrite && (offset_for_prepwrite > 0))
    {
        int copylen = offset_for_prepwrite;
        if (offset_for_prepwrite > sizeof (desp_blobread_prepwrite))
        {
            copylen = sizeof (desp_blobread_prepwrite);
        }
        memcpy (desp_blobread_prepwrite, pBuffer_for_prepwrite, copylen);
    }
    pBuffer_for_prepwrite = NULL; offset_for_prepwrite = 0;
}
else if (ATT_TRANSACTION_MODE_CANCEL == transaction_mode) //cancel write

```

```

    {
        /* The user-defined descriptor of the sixth characteristic receives the
           Cancel write command from the peer. After receiving this command,
           discard the previously stored peer prepare write data and do not
           rewrite the descriptor. */
        if (pbuffer_for_prepwrite)
        {
            pbuffer_for_prepwrite = NULL;
        }
        offset_for_prepwrite = 0;
    }
}
return 0;
}

```

Send Notify

```

uint8_t txrx_server_transfer (uint16_t ul6ConnHdl, uint8_t* pu8Data, uint16_t
length)
{
    // The second characteristic sends notify, which cannot be sent if the value of
    // its Client Config characteristic descriptor is 0
    // Note: The length of each sent notify data cannot exceed (MTU-3)
    if (rx_client_char_cfg)
    {
        return gatts_send_notify (ul6Connhandle, txrx_server_hdl[RX_NOTIFY],
                                pu8Data, length);
    }
    return 0xFE; // notify disable
}

uint8_t txrx_test_cmd_transfer (uint16_t ul6ConnHdl, uint8_t* pu8Data, uint16_t
length)
{
    // The third characteristic sends notify, which cannot be sent if the value of
    // its Client Config characteristic descriptor is 0
    if (txrx_cmd_char_cfg)
    {
        return gatts_send_notify (ul6ConnHdl, txrx_server_hdl[TXRX_CMD], pu8Data,
                                length);
    }
    return 0xFE; // notify disable
}

```

Send Indication

```

uint8_t txrx_test_indicate_data (uint16_t ul6ConnHdl, uint8_t* pu8Data, uint16_t
length)
{
    // The sixth characteristic sends indication, which cannot be sent if the
    // value of its Client Config characteristic descriptor is 0
    // Here is an example for sending 20 bytes of data each time
    if (txrx_indicate_char_cfg)
    {
        return rom_gatts_send_indicate (ul6ConnHdl, txrx_server_hdl[TRX_INDICATE_
                                IDX], pu8Data, length);
    }
    return 0xFE; // indicate disable
}

```

Bluetooth Low Energy GATT Client API Interfaces

API Interfaces

rom_gatt_client_api_init

Function Prototype

```
void rom_gatt_client_api_init (gatt_client_call_back callback);
```

Functional Description

Initialize the protocol stack on the GATT Client.

Parameter

Parameter	Description
callback	Specify a callback function that receives the query result for a service or characteristic, as well as the results of other operations.

Return Value

None.

rom_gatt_client_api_discover_primary_services

Function Prototype

```
uint8_t rom_gatt_client_api_discover_primary_services (uint16_t u16ConnHandle);
```

Functional Description

Query all primary services of the peer device.

Parameter

Parameter	Description
u16ConnHandle	Connection handle, used to distinguish between different connections.

Return Value

status	If the call is successful, return 0. Otherwise, return non-0. A successful call does not mean a successful query.
--------	---

Note

The query result is returned in the callback function registered during protocol stack initialization.

rom_gatt_client_api_discover_primary_services_by_uuid16

Function Prototype

```
uint8_t rom_gatt_client_api_discover_primary_services_by_uuid16 (uint16_t u16ConnHandle,  
uint16_t u16Uuid16);
```

Functional Description

Query the primary service with a specific 16-bit UUID.

Parameter

Parameter	Description
u16ConnHandle	Connection handle, used to distinguish between different connections.
u16Uuid16	Service UUID to be queried.

Return Value

status	If the call is successful, return 0. Otherwise, return non-0.
--------	---

Note

This API is used for querying a single service. The query result is returned in the callback function registered during protocol stack initialization.

rom_gatt_client_api_discover_primary_services_by_uuid128

Function Prototype

```
uint8_t rom_gatt_client_api_discover_primary_services_by_uuid128 (uint16_t u16ConnHandle,
uint8_t* pu8Uuid128);
```

Functional Description

Query the primary service with a specific 128-bit UUID.

Parameter

Parameter	Description
u16ConnHandle	Connection handle, used to distinguish between different connections.
pu8Uuid128	Service UUID to be queried.

Return Value

status	If the call is successful, return 0. Otherwise, return non-0.
--------	---

Note

This API is used for querying a single service. The query result is returned in the callback function registered during protocol stack initialization.

rom_gatt_client_api_discover_characteristics_for_handle_range_by_uuid16

Function Prototype

```
uint8_t rom_gatt_client_api_discover_characteristics_for_handle_range_by_uuid16 (uint16_t
u16ConnHandle, uint16_t u16StartHandle, uint16_t u16EndHandle, uint16_t u16Uuid16);
```

Functional Description

Query characteristics with a specific 16-bit UUID within the specified handle range.

Parameter

Parameter	Description
u16ConnHandle	Connection handle, used to distinguish between different connections.
u16StartHandle	Start handle.
u16EndHandle	End handle.
u16Uuid16	Service UUID to be queried.

Return Value

status	If the call is successful, return 0. Otherwise, return non-0.
--------	---

Note

Query in the range of (u16StartHdl, u16EndHdl). The query result is returned in the callback function registered during protocol stack initialization.

rom_gatt_client_api_discover_characteristics_for_handle_range_by_uuid128

Function Prototype

```
uint8_t rom_gatt_client_api_discover_characteristics_for_handle_range_by_uuid128 (uint16_t u16ConnHandle, uint16_t u16StartHandle, uint16_t u16EndHandle, uint8_t* pu8Uuid128);
```

Functional Description

Query characteristics with a specific 128-bit UUID within the specified handle range.

Parameter

Parameter	Description
u16ConnHandle	Connection handle, used to distinguish between different connections.
u16StartHandle	Start handle.
u16EndHandle	End handle.
pu8Uuid128	Specified 128-bit UUID.

Return Value

status	If the call is successful, return 0. Otherwise, return non-0.
--------	---

Note

Query in the range (u16StartHdl, u16EndHdl). The query result is returned in the callback function registered during protocol stack initialization.

rom_gatt_client_api_discover_characteristic_descriptors

Function Prototype

```
uint8_t rom_gatt_client_api_discover_characteristic_descriptors (uint16_t u16ConnHandle, gatt_client_characteristic_t* pstCharacteristic);
```

Functional Description

Query the characteristic descriptor for the specified characteristic.

Parameter

Parameter	Description
u16ConnHandle	Connection handle, used to distinguish between different connections.
pstCharacteristic	Specified characteristic.

Return Value

status	If the call is successful, return 0. Otherwise, return non-0.
--------	---

Note

The query result is returned in the callback function registered during protocol stack initialization.

rom_gatt_client_api_discover_characteristics_for_service

Function Prototype

```
uint8_t rom_gatt_client_api_discover_characteristics_for_service (uint16_t u16ConnHandle, gatt_client_service_t* pstService);
```

Functional Description

Query all included characteristics for the specified service.

Parameter

Parameter	Description
u16ConnHandle	Connection handle, used to distinguish between different connections.
pstService	Specified service.

Return Value

status	If the call is successful, return 0. Otherwise, return non-0.
--------	---

Note

The query result is returned in the callback function registered during protocol stack initialization.

rom_gatt_client_api_find_included_services_for_service

Function Prototype

```
uint8_t rom_gatt_client_api_find_included_services_for_service (uint16_t u16ConnHandle, gatt_client_service_t* pstService);
```

Functional Description

Query the included services for the specified service.

Parameter

Parameter	Description
u16ConnHandle	Connection handle, used to distinguish between different connections.
pstService	Specified service.

Return Value

status	If the call is successful, return 0. Otherwise, return non-0.
--------	---

Note

The query result is returned in the callback function registered during protocol stack initialization.

rom_gatt_client_api_write_client_characteristic_configuration

Function Prototype

```
uint8_t rom_gatt_client_api_write_client_characteristic_configuration (uint16_t u16ConnHandle, gatt_client_characteristic_t* pstCharacteristic, uint16_t configuration);
```

Functional Description

Used for the Client to configure the Characteristic CCCD (client_characteristic_configuration descriptor) of the Server.

Parameter

Parameter	Description
u16ConnHandle	Connection handle, used to distinguish between different connections.
pstCharacteristic	Specified characteristic.
configuration	Setting value. 0: Disable notify and indicate 1: Enable notify and disable indicate 2: Enable indicate and disable notify 3: Enable indicate and enable notify

Return Value

status	If the call is successful, return 0. Otherwise, return non-0.
--------	---

Note

Whether to send notify is determined by the server.

rom_gatt_client_api_read_long_value_of_characteristic_using_value_handle_with_offset

Function Prototype

```
uint8_t rom_gatt_client_api_read_long_value_of_characteristic_using_value_handle_with_offset(
    uint16_t u16ConnHandle, uint16_t characteristic_value_handle, uint16_t u16Offset);
```

Functional Description

Read long characteristic value.

Parameter

Parameter	Description
u16ConnHandle	Connection handle, used to distinguish between different connections.
characteristic_value_handle	Characteristic handle to be read.
u16Offset	Characteristic value offset, which is cleared to 0 to indicate that it is read from the beginning.

Return Value

status	If the call is successful, return 0. Otherwise, return non-0.
--------	---

Note

If the length of the characteristic value to be read is greater than MTU-3, its contents will be returned several times in the callback function registered during protocol stack initialization until the read is completed.

rom_gatt_client_api_cancel_write

Function Prototype

```
uint8_t rom_gatt_client_api_cancel_write (uint16_t u16ConnHandle);
```

Functional Description

Before this operation, there will be an operation that prepares the characteristic value to be modified using Prepare write. Calling this API will discard the previously stored content.

Parameter

Parameter	Description
u16ConnHandle	Connection handle, used to distinguish between different connections.

Return Value

status	If the call is successful, return 0. Otherwise, return non-0.
--------	---

Note

The operation result is returned in the callback function registered during protocol stack initialization.

rom_gatt_client_api_execute_write

Function Prototype

```
uint8_t rom_gatt_client_api_execute_write (uint16_t u16ConnHandle);
```

Functional Description

Before this operation, there will be an operation that prepares the characteristic value to be modified using Prepare write. Calling this API will modify the characteristic value using the previously stored content.

Parameter

Parameter	Description
u16ConnHandle	Connection handle, used to distinguish between different connections.

Return Value

status	If the call is successful, return 0. Otherwise, return non-0.
--------	---

Note

The operation result is returned in the callback function registered during protocol stack initialization.

rom_gatt_client_api_prepare_write

Function Prototype

```
uint8_t rom_gatt_client_api_prepare_write (uint16_t u16ConnHandle, uint16_t attribute_handle,
uint16_t u16Offset, uint16_t u16Len, uint8_t* pu8Data);
```

Functional Description

Prepare the characteristic value to be modified using Prepare write. This mode is used when the characteristic value is long. This operation does not immediately modify the characteristic value, it just temporarily stores the content to be written.

Parameter

Parameter	Description
u16ConnHandle	Connection handle, used to distinguish between different connections.
attribute_handle	Characteristic handle.
u16Offset	Start offset of Prepare write.
u16Len	Content length to be written.
pu8Data	Content to be written.

Return Value

status	If the call is successful, return 0. Otherwise, return non-0.
--------	---

Note

The operation result is returned in the callback function registered during protocol stack initialization. The returned data contains the transmission data received by the peer. This data can be compared with the prepare write data to check whether the peer receives the data correctly.

rom_gatt_client_api_reliable_write_long_value_of_characteristic

Function Prototype

```
uint8_t rom_gatt_client_api_reliable_write_long_value_of_characteristic (uint16_t u16ConnHandle,
uint16_t u16ValueHandle, uint16_t value_length, uint8_t* pu8Value);
```

Functional Description

Used to modify long characteristic values. The specific implementation is to Prepare write several times, and finally to execute write, which is equivalent to using a combination of rom_gatt_client_api_discover_characteristics_for_service and rom_gatt_client_api_discover_characteristics_for_handle_range_by_uuid16.

Parameter

Parameter	Description
u16ConnHandle	Connection handle, used to distinguish between different connections.
u16ValueHandle	Characteristic handle.
value_length	Content length to be written.
pu8Value	Content to be written.

Return Value

status	If the call is successful, return 0. Otherwise, return non-0.
--------	---

Note

The operation result is returned in the callback function registered during protocol stack initialization. This API is basically the same as rom_gatt_client_api_write_long_value_of_characteristic. The difference is that the API checks whether the peer receives the data correctly during the Prepare write process. If the peer receives it incorrectly, the transmission is stopped and an error is returned. The rom_gatt_client_api_write_long_value_of_characteristic does not detect whether the peer has received the data correctly.

rom_gatt_client_api_write_long_value_of_characteristic

Function Prototype

```
uint8_t rom_gatt_client_api_write_long_value_of_characteristic (uint16_t u16ConnHandle, uint16_t
u16ValueHandle, uint16_t value_length, uint8_t* pu8Value);
```

Functional Description

Used to modify long characteristic values. The specific implementation is to Prepare write several times, and finally to execute write, which is equivalent to a combination of rom_gatt_client_api_discover_characteristics_for_service and rom_gatt_client_api_discover_characteristics_for_handle_range_by_uuid16.

Parameter

Parameter	Description
u16ConnHandle	Connection handle, used to distinguish between different connections.
u16ValueHandle	Characteristic handle.
value_length	Content length to be written.
pu8Value	Content to be written.

Return Value

status	If the call is successful, return 0. Otherwise, return non-0.
--------	---

Note

The operation result is returned in the callback function registered during protocol stack initialization.

rom_gatt_client_api_write_value_of_characteristic_without_response

Function Prototype

```
uint8_t rom_gatt_client_api_write_value_of_characteristic_without_response (uint16_t u16ConnHandle, uint16_t u16ValueHandle, uint16_t value_length, uint8_t* pu8Value);
```

Functional Description

Modify the characteristic value for the specified handle in an unreliable write mode.

Parameter

Parameter	Description
u16ConnHandle	Connection handle, used to distinguish between different connections.
u16ValueHandle	Characteristic handle.
value_length	Content length to be written.
pu8Value	Content to be written.

Return Value

status	If the call is successful, return 0. Otherwise, return non-0.
--------	---

Note

This operation does not return any result in the callback function.

rom_gatt_client_api_write_value_of_characteristic

Function Prototype

```
uint8_t rom_gatt_client_api_write_value_of_characteristic (uint16_t u16ConnHandle, uint16_t u16ValueHandle, uint16_t value_length, uint8_t* pu8Data);
```

Functional Description

Modify the characteristic value for the specified handle in a reliable write mode.

Parameter

Parameter	Description
u16ConnHandle	Connection handle, used to distinguish between different connections.
u16ValueHandle	Characteristic handle.
value_length	Content length to be written.
pu8Value	Content to be written.

Return Value

status	If the call is successful, return 0. Otherwise, return non-0.
--------	---

Note

The operation result is returned in the callback function registered during protocol stack initialization.

rom_gatt_client_api_read_value_of_characteristic_using_value_handle

Function Prototype

```
uint8_t rom_gatt_client_api_read_value_of_characteristic_using_value_handle (uint16_t u16ConnHandle, uint16_t u16ValueHandle);
```

Functional Description

Read the characteristic value for the specified handle.

Parameter

Parameter	Description
u16ConnHandle	Connection handle, used to distinguish between different connections.
u16ValueHandle	Characteristic handle.

Return Value

status	If the call is successful, return 0. Otherwise, return non-0.
--------	---

Note

The read content is returned in the callback function registered during protocol stack initialization.

rom_gatt_client_api_read_value_of_characteristics_by_uuid16

Function Prototype

```
uint8_t rom_gatt_client_api_read_value_of_characteristics_by_uuid16 (uint16_t u16ConnHandle, uint16_t u16StartHandle, uint16_t u16EndHandle, uint16_t u16Uuid16);
```

Functional Description

Query and read the characteristic values for the specified 16-bit UUID between u16StartHandle and u16EndHandle.

Parameter

Parameter	Description
u16ConnHandle	Connection handle, used to distinguish between different connections.
u16StartHandle	Start handle of the query.
u16EndHandle	End handle of the query.
u16Uuid16	Specified UUID.

Return Value

status	If the call is successful, return 0. Otherwise, return non-0.
--------	---

Note

The read contents is returned in the callback function registered during protocol stack initialization.

rom_gatt_client_api_read_value_of_characteristics_by_uuid128

Function Prototype

```
uint8_t rom_gatt_client_api_read_value_of_characteristics_by_uuid128 (uint16_t u16ConnHandle,  
uint16_t u16StartHandle, uint16_t u16EndHandle, uint8_t* pu8Uuid128);
```

Functional Description

Query and read the characteristic values for the specified 128-bit UUID between u16StartHandle and u16EndHandle.

Parameter

Parameter	Description
u16ConnHandle	Connection handle, used to distinguish between different connections.
u16StartHandle	Start handle of the query.
u16EndHandle	End handle of the query.
pu8Uuid128	Specified UUID.

Return Value

status	If the call is successful, return 0. Otherwise, return non-0.
--------	---

Note

The read content is returned in the callback function registered during protocol stack initialization.

rom_gatt_client_api_write_characteristic_descriptor_using_descriptor_handle

Function Prototype

```
uint8_t rom_gatt_client_api_write_characteristic_descriptor_using_descriptor_handle (uint16_t  
u16ConnHandle, uint16_t u16DescriptorHandle, uint16_t u16Len, uint8_t* pu8Data);
```

Functional Description

Rewrite the characteristic descriptor.

Parameter

Parameter	Description
u16ConnHandle	Connection handle, used to distinguish between different connections.
u16DescriptorHandle	Descriptor handle.
u16Len	Content length to be rewritten.
pu8Data	Content to be rewritten.

Return Value

status	If the call is successful, return 0. Otherwise, return non-0.
--------	---

Note

The operation result is returned in the callback function registered during protocol stack initialization.

rom_gatt_client_api_mtu_exchange

Function Prototype

```
uint8_t rom_gatt_client_api_mtu_exchange (uint16_t conn_handle, uint16_t u16Mtu);
```

Functional Description

Negotiate with the peer to change the MTU.

Parameter

Parameter	Description
u16ConnHandle	Connection handle, used to distinguish between different connections.
u16Mtu	MTU value supported by the local.

Return Value

status	If the call is successful, return 0. Otherwise, return non-0.
--------	---

Note

The operation result is returned in the callback function registered during protocol stack initialization. Actually return an MTU that is supported by both parties.

Callback Function

Function Prototype

```
uint32_t (*gatt_client_call_back) (uint16_t u16ConnHandle, uint16_t u16OpCode, uint16_t u16ErrCode, uint8_t* pu8Buf, uint16_t u16Len);
```

Description

Used to return the operation result of the Client API.

Parameter

Parameter	Description
u16OpCode	Opcode, used to distinguish which operation result it is. Refer to the opcode definition below.
u16ErrCode	The lower 8 bits indicate an error code, and whether bit15 is 1 indicates whether the operation is completed.
pu8Buf	Returned data, which has different meanings for different operation data.
u16Len	Returned data length.

Return Value

status	Ignore, this function is called internally by the protocol stack.
--------	---

Note

With the exception of the unreliable write `gatt_client_write_value_of_characteristic_without_response`, other Client APIs that have been successfully called will return their operation results using this callback function. Opcode definition and meaning:

```
typedef enum
{
    GATTC_USER_NOTIFICATION, // The peer device sends notify data
    GATTC_USER_INDICATION,   // The peer device sends indication data
    GATTC_USER_MTU_EXCHG,    // Modified MTU result
    GATTC_USER_INCLUDE_SERVICES_QUERY, // Query include services
    GATTC_USER_PRIMARY_SERVICES_QUERY_ALL, // Query all primary services
    GATTC_USER_PRIMARY_SERVICES_QUERY_ONE, // Query a single service
    GATTC_USER_CHARACTERISTICS_QUERY_ALL, // Query all characteristics included
                                        // in a service
    GATTC_USER_CHARACTERISTICS_QUERY_ONE_BY_UUID, // Query characteristics for
                                                // the specified UUID
    GATTC_USER_CHARACTERISTICS_DESCRIPTOR_QUERY, // Query characteristic
                                                // descriptors
    GATTC_USER_CHARACTERISTICS_WRITE_RESULT, // Modify characteristic
                                                // values using reliable write
    GATTC_USER_CHARACTERISTICS_READ_BY_HANDLE_RESULT, // Read characteristic
                                                // values for the specified
                                                // handle
    GATTC_USER_CHARACTERISTICS_READ_BY_UUID_RESULT, // Read characteristic values
                                                // for the specified UUID
    GATTC_USER_CHARACTERISTICS_BLOB_READ_BY_HANDLE_RESULT, // Read long
                                                // characteristic
                                                // values
    GATTC_USER_PREPARE_WRITE_CHAR_RESULT, // Prepare characteristic values to be
                                                // written using Prepare write
    GATTC_USER_PREPARE_WRITE_EXECUTE_CHAR_RESULT, // Execut write characteristic
                                                // value operation
    GATTC_USER_PREPARE_WRITE_CANCEL_CHAR_RESULT, // Cancel write characteristic
                                                // value operation
    GATTC_USER_WRITE_LONG_CHAR_RESULT, // Write long characteristic values
    GATTC_USER_RELIABLE_WRITE_LONG_CHAR_RESULT, // Write long characteristic
                                                // values using reliable write
    GATTC_USER_CHARACTERISTICS_DESCRIP_WRITE_RESULT, // Write characteristic
                                                // descriptors
} user_client_state;
```

Structure Definition

Service Information Structure, includes the start and end handle, and UUID of the service, as shown below. Among them, the UUID may be 16 bits or 128 bits, which are stored in `uuid16` and `uuid128` respectively. If it is 128 bits, `uuid16` is 0, otherwise, `uuid128` is all 0.

```
typedef struct
{
    uint16_t start_group_handle;
    uint16_t end_group_handle;
    uint16_t uuid16;
    uint8_t uuid128[16];
} gatt_client_service_t;
```

Characteristic Information Structure, includes the handle, property, UUID and other information of the characteristic. Each characteristic definition must be preceded by a characteristic declaration, followed by the characteristic value definition, and possibly followed by the characteristic descriptor definition.

In the structure, `start_handle` is the handle that points to the characteristic declaration, `value_handle` is the handle that points to the characteristic value, and `end_handle` is equal to `value_handle` if there is no descriptor and it the handle of the last descriptor if there are descriptors.

```
typedef struct
{
    uint16_t start_handle;
    uint16_t value_handle;
    uint16_t end_handle;
    uint16_t properties;
    uint16_t uuid16;
    uint8_t uuid128[16];
} gatt_client_characteristic_t;
```

Characteristic Descriptor Information Structure, includes the handle and UUID of the descriptor, as shown below.

```
typedef struct
{
    uint16_t handle;
    uint16_t uuid16;
    uint8_t uuid128[16];
} gatt_client_characteristic_descriptor_t;
```

Include Service Information Structure, is as follows, where `include_handle` is the handle of the current include service declaration, `include_serv_start_group_handle` and `include_serv_end_group_handle` are the start and end handle of the include service, and the last two parameters are its UUID.

```
typedef struct
{
    uint16_t include_handle;
    uint16_t include_serv_start_group_handle;
    uint16_t include_serv_end_group_handle;
    uint16_t uuid16;
    uint8_t uuid128[16];
} gatt_client_include_service_t;
```

Error Code

1. Error code:

#define ATT_ERROR_INVALID_HANDLE	0x01
#define ATT_ERROR_READ_NOT_PERMITTED	0x02
#define ATT_ERROR_WRITE_NOT_PERMITTED	0x03
#define ATT_ERROR_INVALID_PDU	0x04
#define ATT_ERROR_INSUFFICIENT_AUTHENTICATION	0x05
#define ATT_ERROR_REQUEST_NOT_SUPPORTED	0x06
#define ATT_ERROR_INVALID_OFFSET	0x07
#define ATT_ERROR_INSUFFICIENT_AUTHORIZATION	0x08
#define ATT_ERROR_PREPARE_QUEUE_FULL	0x09
#define ATT_ERROR_ATTRIBUTE_NOT_FOUND	0x0A
#define ATT_ERROR_ATTRIBUTE_NOT_LONG	0x0B
#define ATT_ERROR_INSUFFICIENT_ENCRYPTION_KEY_SIZE	0x0C
#define ATT_ERROR_INVALID_ATTRIBUTE_VALUE_LENGTH	0x0D
#define ATT_ERROR_UNLIKELY_ERROR	0x0E
#define ATT_ERROR_INSUFFICIENT_ENCRYPTION	0x0F

```
#define ATT_ERROR_UNSUPPORTED_GROUP_TYPE          0x10
#define ATT_ERROR_INSUFFICIENT_RESOURCES          0x11
#define ATT_ERROR_HCI_DISCONNECT_RECEIVED         0x1F
#define ATT_ERROR_BONDING_INFORMATION_MISSING     0x70
#define ATT_ERROR_DATA_MISMATCH                   0x7E
#define ATT_ERROR_TIMEOUT                         0x7F
```

2. The common error codes are as follows:

ATT_ERROR_INVALID_HANDLE: The handle used is invalid.

ATT_ERROR_READ_NOT_PERMITTED: Execute a read operation on a characteristic without the read preoperty.

ATT_ERROR_WRITE_NOT_PERMITTED: Execute a write operation on a characteristic without the write preoperty.

ATT_ERROR_INSUFFICIENT_AUTHENTICATION: Insufficient authentication. The connection needs to be encrypted before it can be accessed.

ATT_ERROR_INVALID_OFFSET: The offset is invalid.

ATT_ERROR_ATTRIBUTE_NOT_FOUND: The characteristic is not found.

ATT_ERROR_PREPARE_QUEUE_FULL: The queue for prepare write is full and cannot proceed with prepare write.

ATT_ERROR_DATA_MISMATCH: Indicates a data transmission failure for long characteristic value write operation.

Example Code

Initialize and Register Callback Function

First, initialize the protocol stack and register the callback function, where `ble_gatt_client_call_back` is the callback function:

```
gatt_client_init (ble_gatt_client_call_back);
```

Query Service

After entering the connection, the following code requires to be executed to enable the query of all the primary services:

```
gatt_client_discover_primary_services (u16ConnHandle);
```

The query result is returned in the callback function with the opcode of `GATTC_USER_PRIMARY_SERVICES_QUERY_ALL`. Each time a primary service is queried, a callback function is called to return the service information. Since the peer may include multiple primary services, the callback function may be called multiple times. At the end of the query, it will also be called once. At this time, `errcode` bit15 is 1, indicating that the query service is completed.

```
uint32 ble_gatt_client_call_back(uint16 connHdl, uint16 opcode, uint16 errcode,
uint8 *buf, uint16 len)
{
    uint16_t isClientOver = (errcode & 0x8000); errcode &= 0xff;
    int ret = 0;
    switch (opcode)
    {
        case GATTC_USER_PRIMARY_SERVICES_QUERY_ALL:
        {
            // Query not completed, return the queried service information
            if (is_client_over == 0)
```

```

    {
        if (buf && (errcode == 0))
        {
            // The returned service information is stored in buf,
            // which is actually a service structure, including the
            // start and end handle, and UUID of the service
            gatt_client_service_t *pser = (gatt_client_service_t *)buf;
            // This is a demo code for storing the service information
            ret = ble_central_add_a_service((gatt_client_service_t *)buf);
            if (TX_RX_SERVICE_UUID == pser->uuid16)
            {
                // If it is a transparent service, store its information
                // and use it later for transparent testing
                memcpy(&g_trx_ser, (void *)pser, sizeof(gatt_client_service_t));
            }
        }
    }
    else
    {
        // Query completed
        PRINTF("discover_primary_services over");
    }
    break;
    .....
}
}
}

```

If it requires to query only a single service, such as the GAP service with a service UUID of 0x1800, the following code can be executed:

```
gatt_client_discover_primary_services_by_uuid16 (0, 0x1800);
```

The query result is returned in the callback function with the opcode of GATTC_USER_PRIMARY_SERVICES_QUERY_ONE. The following code is in the callback function ble_gatt_client_callback.

```

case GATTC_USER_PRIMARY_SERVICES_QUERY_ONE:
{
    if ((errcode == 0) && buf)
    {
        // The query succeeds, the returned service information is stored in buf,
        // which is actually a service structure, including the
        // start and end handle, and UUID of the service
        gatt_client_service_t *pser = (gatt_client_service_t *)buf;
        gSer_start_handle = pser->start_group_handle;
        gSer_end_handle = pser->end_group_handle;
    }
    else if (ATT_ERROR_ATTRIBUTE_NOT_FOUND == errcode)
    {
        // The query fails, service not found
        PRINTF("not found service");
    }
    break;
}
}

```

Query the include service. If it requires to query the services included in a transparent service, the following code can be executed, where g_trx_ser is the structure that previously stores the transparent service information:

```
gatt_client_find_included_services_for_service (0, &g_trx_ser);
```

The query result is returned in the callback function with the opcode of GATTC_USER_INCLUDE_SERVICES_QUERY, as follows:

```
case GATTC_USER_INCLUDE_SERVICES_QUERY:
{
    if (buf && (errcode == 0))
    {
        // The include service is queried, buf contains its information,
        // which is actually a include service structure.
        gatt_client_include_service_t *pinclude_ser = (gatt_client_include_
            service_t *)buf;
    }
    if (isClientOver)
    {
        // The query is completed
        PRINTF("find include service over");
    }
    break;
}
```

Query Characteristic

To query all the characteristics included in a service, the following code can be executed:

```
gatt_client_discover_characteristics_for_service (0, &g_trx_ser);
```

Where g_trx_ser is the structure that previously stores the transparent service, it will query all the characteristics contained in the transparent service. The query result is returned in the callback function with the opcode of GATTC_USER_CHARACTERISTICS_QUERY_ALL, as shown below. This callback function is called once every time a characteristic is found. As the service may contain multiple characteristics, it may be called multiple times:

```
case GATTC_USER_CHARACTERISTICS_QUERY_ALL:
{
    if (isClientOver == 0)
    {
        if (errcode == 0 && (buf != NULL))
        {
            // A characteristic is found, buf is actually a characteristic information
            // structure, which contains the information of the characteristic
            gatt_client_characteristic_t *pchr = (gatt_client_characteristic_t *)buf;
            ret = ble_central_add_a_characteristic((gatt_client_characteristic_t *)buf);
            if (RX_CUS_UUID == pchr->uuid16)
            {
                // Characteristic used for uplink in the transparent service,
                // save its information for later transparent testing
                memcpy(&g_rx_char, (void *)pchr, sizeof(gatt_client_characteristic_t));
            }
            else if (TX_CUS_UUID == pchr->uuid16)
            {
                // Characteristic used for downlink in the transparent service,
                // save its information for later transparent testing
                memcpy(&g_tx_char, (void *)pchr, sizeof(gatt_client_characteristic_t));
            }
            else if (TRX_CMD_UUID == pchr->uuid16)
            {
                // Characteristic used to transmit commands in the transparent service,
                // save its information for later transparent testing
                memcpy(&g_cmdrx_char, (void *)pchr, sizeof(gatt_client_characteristic_
                    t));
            }
        }
    }
}
```

```

else if (TRX_BLOBREAD_PREPWRITE_TEST_UUID == pchr->uuid16)
{
    // Characteristic used to test the read and write of
    // long characteristics in the transparent service,
    // save its information for later transparent testing
    memcpy(&g_blobread_preparewrite_char, (void *)pchr, sizeof(gatt_
        client_characteristic_t));
}
else if (TRX_INDICATE_UUID == pchr->uuid16)
{
    // Characteristic used to test indication in the transparent service,
    // save its information for later transparent testing
    memcpy(&g_indicate_char, (void *)pchr, sizeof(gatt_client_
        characteristic_t));
}
}
else
{
    // The characteristic query is completed
}
break;
}

```

Query Characteristic Descriptor

To query all descriptors contained in the characteristic for data uplink in the transparent service, the following code can be executed:

```
gatt_client_discover_characteristics_for_service (0, & g_rx_char);
```

Where `g_rx_char` is the structure that previously stores the characteristic, it will query the descriptors contained in the characteristic. The query result is returned in the callback function with the opcode of `GATTC_USER_CHARACTERISTICS_DESCRIPTOR_QUERY`, as shown below. This callback function is called once every time a descriptor is found. As the characteristic may contain multiple descriptors, it may be called multiple times:

```

case GATTC_USER_CHARACTERISTICS_DESCRIPTOR_QUERY:
{
    if (isClientOver == 0)
    {
        if (errcode == 0 && (buf != NULL))
        {
            // Return a characteristic descriptor information
            gatt_client_characteristic_descriptor_t *per = (gatt_client_
                characteristic_descriptor_t *)buf;
            if (GATT_CLIENT_CHAR_CFG_UUID == pchr->uuid16)
            {
                // Characteristic Client Config descriptor,
                // save its handle for later testing
                g_tx_cc_hdl = pchr->handle;
            }
        }
    }
    else
    {
        // The query is completed
        Printf("found descriptor over");
    }
}
}

```

Characteristic Value Read

The characteristic to be read must have a read property (ATT_PROPERTY_READ) before it can be read. There are two modes to read characteristic values:

Mode 1: Reading by specifying a handle and execute the following code. This is a normal read mode. If the characteristic value length exceeds MTU-3, the length of the characteristic value returned by this read mode is MTU-3:

```
gatt_client_read_value_of_characteristic_using_value_handle(0,g_blobread_
preparewrite_char.handle);
```

The second parameter is the characteristic handle, and g_blobread_preparewrite_char is the structure that stores the characteristic. The execution result is returned in the callback function with the opcode of GATT_USER_CHARACTERISTICS_READ_BY_HANDLE_RESULT, as shown below. This read mode only has one callback with the read content.

```
case GATT_USER_CHARACTERISTICS_READ_BY_HANDLE_RESULT:
{
    if (isClientOver == 0)
    {
        if (buf && (errcode == 0))
        {
            // buf carries the read data, with a length of len
            PRINTF("read by handle[%d]:", len);
        }
    }
    else
    {
        // If the read fails, the callback function is called only once, carrying an
        // error code. If the read succeeds, the callback function is called only
        // twice, the first time carrying the read data, and the second time
        // (errcode=0) indicating that the read is completed.
        PRINTF("read by handle over [%x]\n", errcode);
    }
    break;
}
```

Mode 2: Reading by specifying a UUID. This is also a normal read mode. If the characteristic value length exceeds MTU-3, the length of the characteristic value returned by this read mode is also MTU-3:

```
gatt_client_read_value_of_characteristics_by_uuid16(0,1,0xffff, g_blobread_
preparewrite_char.uuid16);
```

The fourth parameter is the characteristic UUID, and g_blobread_preparewrite_char is the structure that stores the characteristic. The second and third parameters are the start and end handle of the query, respectively. In this mode, a characteristic with the UUID equal to the fourth parameter is queried in the handle range determined by the second and third parameters, and the characteristic value is read and returned. The execution result is returned in the callback function with the opcode of GATT_USER_CHARACTERISTICS_READ_BY_UUID_RESULT, as shown below. This read mode only has one callback with the read content.

```
case GATT_USER_CHARACTERISTICS_READ_BY_UUID_RESULT:
{
    if (isClientOver == 0)
    {
        if (buf && (errcode == 0))
        {
            // buf carries the read data, with a length of len
            printf("read by uuid[%d]:", len);
        }
    }
}
```



```

    }
}
else
{
    // If the read fails, the callback function is called only once, carrying
    // an error code. If the read succeeds, the callback function is called only
    // twice, the first time carrying the read data, and the second time
    // (errcode=0) indicating that the read is completed.
    printf("read by uuid over [%x]\n", errcode);
}
break;
}

```

Long Characteristic Value Read

Using the read modes described in the previous section cannot read all the contents of long characteristic values (the characteristic value length exceeds MTU-3). If it requires to read all the contents of long characteristic values, the long characteristic value read mode must be used. Take the characteristics used in the previous section as an example, execute the following code:

```

gatt_client_read_long_value_of_characteristic_using_value_handle_with_offset
(0, g_blobread_preparewrite_char.uuid16, 0);

```

The second parameter is the characteristic handle, and `g_blobread_preparewrite_char` is the structure that stores the characteristic. The third parameter is 0, which indicates that it is read from the beginning. The execution result is returned in the callback function with the opcode of `GATTC_USER_CHARACTERISTICS_BLOB_READ_BY_HANDLE_RESULT`, as shown below. This read mode returns the characteristic value through multiple callbacks until the read is completed:

```

case GATTC_USER_CHARACTERISTICS_BLOB_READ_BY_HANDLE_RESULT:
{
    if (isClientOver == 0)
    {
        if (buf && (errcode == 0))
        {
            // buf carries the read data, for long characteristic values, which will be
            // returned here in sequence, such as the first time is the first 20
            // bytes, the second time is 20th to 40th bytes,...
            PRINTF("blob read[%d]:", len);
        }
    }
    else
    {
        // The read is completed successfully or ended due to a read error.
        PRINTF("blob read over [%x]\n", errcode);
    }
    break;
}

```

Characteristic Value Write Operation

There are two modes of write operations, one is a reliable write mode and the other is an unreliable write mode.

1. Reliable write mode: The characteristic must have the ATT_PROPERTY_WRITE property to use this mode. For example, to rewrite the characteristic value of a characteristic that tests long characteristic value read and write in the transparent service, the following code can be executed:

```
gatt_client_write_value_of_characteristic(0,g_blobread_preparewrite_char.  
handle, len, pdata);
```

The second parameter is the characteristic handle, the third and fourth parameters specify the data to be written and its length. The execution result is returned in the callback function with the opcode of GATTC_USER_CHARACTERISTICS_WRITE_RESULT, as shown below. This operation only has one callback, which returns whether the write is successful.

```
case GATTC_USER_CHARACTERISTICS_WRITE_RESULT:  
{  
    // errcode indicates whether the write operation is successful. 0  
    // indicates success.  
    PRINTF("write over [%x]\n", errcode); // write over;  
    break;  
}
```

2. Unreliable write mode: The characteristic must have the ATT_PROPERTY_WRITE_WITHOUT_RESPONSE property to use this mode. For example, to rewrite the characteristic value of the data downlink characteristic in the transparent service, the following code can be executed:

```
gatt_client_write_value_of_characteristic_without_response (0, g_tx_char.  
handle, len, pdata);
```

Where the second parameter is the characteristic handle, and g_tx_char is the structure that previously stores the data downlink characteristic information. The third and fourth parameters specify the data to be written and its length. This operation does not return any execution result, and it cannot determine whether the write is successful, so it is an unreliable write operation.

In addition, the g_tx_char characteristic does not have the ATT_PROPERTY_WRITE property, so it cannot be operated in a reliable write mode. Similarly, g_blobread_preparewrite_char, that does not have the ATT_PROPERTY_WRITE_WITHOUT_RESPONSE property, cannot be operated in an unreliable write mode.

Long Characteristic Value Write Operation

There are three modes for writing long characteristic values.

1. First mode: Prepare write multiple times and then execute write. Taking the characteristic for testing long characteristic read and write in the transparent service as an example, the current MTU is 23, if it requires to rewrite the content of a 50-byte array, and the following code needs to be executed:

Prepare write the first 20 bytes first. As the MTU is 23, a maximum of 20 bytes can be written each time. The second parameter is the characteristic handle. The third parameter is offset, indicating the offset of the current data write, with 0 indicating starting from the beginning. The third and fourth parameters are the written data and its length.

```
gatt_client_prepare_write (0, g_blobread_preparewrite_char.handle, 0, 20,  
pdata);
```

Prepare write 20 bytes for the second time and change the offset parameter to 20. Therefore it needs to write with an offset of 20 bytes.

```
gatt_client_prepare_write (0, g_blobread_preparewrite_char.handle, 20, 20,
pdata+20);
```

Prepare write 10 bytes for the third time and change the offset parameter to 40. Therefore, it needs to write with an offset of 40 bytes, as 40 bytes have already been written.

```
gatt_client_prepare_write (0, g_blobread_preparewrite_char.handle, 40, 10,
pdata+40);
```

Finally, execute the write operation to rewrite the previously prepared data into the characteristic value. Because no data is written into the characteristic value during the previous three prepare write operations, the peer device stores the data after receiving prepare write data and can only write it until the execute_write is performed.

```
gatt_client_execute_write (0);
```

In addition, if the characteristic value needs to be rewritten at the end, the following code can be executed to discard the previously prepared data to be written:

```
gatt_client_cancel_write (0);
```

These three operations will return the corresponding results in the callback function, as follows:

```
case GATTC_USER_PREPARE_WRITE_CHAR_RESULT:
{
    // The return for the prepare write operation
    if (buf && (errcode == 0))
    {
        // If the operation succeeds, the data received by the peer is returned,
        // which can be compared with the sent data to determine
        // whether the data is successfully transmitted.
        PRINTF("prepare wr [len%d]:", len);
    }
    if (isClientOver)
    {
        // The operation succeeds, or an error code is returned if the operation
        // fails
        PRINTF("prepare wr over [%x]\n", errcode);
    }
    break;
}
case GATTC_USER_PREPARE_WRITE_EXCUTE_CHAR_RESULT:
{
    // The return for the write operation
    if (isClientOver)
    {
        // The operation succeeds, or an error code is returned if the operation
        // fails
        PRINTF("execute write over [%x]\n", errcode);
    }
    break;
}
case GATTC_USER_PREPARE_WRITE_CANCEL_CHAR_RESULT:
{
    // The return for the cancel write operation
    if (isClientOver)
    {
        // The operation succeeds, or an error code is returned if the operation
        // fails
        PRINTF("prepare write cancel over [%x]\n", errcode);
    }
    break;
}
```

2. Second mode:

```
gatt_client_write_long_value_of_characteristic(0,g_blobread_preparewrite_
char.handle, 0,50, pdata);
```

3. Third mode:

```
gatt_client_reliable_write_long_value_of_characteristic(0,g_blobread_
preparewrite_char.handle, 0,50, pdata);
```

These two operations are also implemented by the protocol stack using multiple prepare writes and an execute write. The difference is that for the third mode, the protocol stack checks whether the data is transmitted successfully in each prepare write response message. If the data is incorrect, the transmission will be terminated and an error will be returned. The second mode does not check whether the data transmission is successful. Their execution results are returned in the callback function.

```
case GATTC_USER_WRITE_LONG_CHAR_RESULT:
{
    // The return for the second mode
    if (isClientOver)
    {
        // The operation succeeds, or an error code is returned if the operation
        // fails
        PRINTF("write long char over [%x]\n", errcode);
    }
    break;
}
case GATTC_USER_RELIABLE_WRITE_LONG_CHAR_RESULT:
{
    // The return for the third mode
    if (ATT_ERROR_DATA_MISMATCH == errcode)
    {
        // If a data transmission error is detected during the prepare write
        // process, the error code is returned
        PRINTF("reliable write long char error cause _DATA_MISMATCH \n");
    }
    if (isClientOver)
    {
        // The operation succeeds, or an error code is returned if the operation
        // fails
        PRINTF("reliable write long char over [%x]\n", errcode);
    }
    break;
}
```

Also note that if the characteristic does not have the ATT_PROPERTY_WRITE property, it cannot be written in these three modes.

Modify Characteristic Descriptor

For transparent testing data uplink, it needs to modify the Client Config descriptor of the uplink characteristic to enable it to send notifications. The following code requires to be executed:

```
// This characteristic has the notification property, and the descriptor is
// configured to 0x0001 to enable notification. For characteristics with indication
// property, the descriptor value should be configured to 0x0002.
unsigned char data[2] = {1, 0};
gatt_client_write_characteristic_descriptor_using_descriptor_handle(0, g_tx_
cc_hdl, 2, data);
```

`g_tx_cc_hdl` is the handle of the Client Config descriptor for the uplink characteristic saved previously. The operation result is returned in the callback function. This operation only has one callback return, as follows:

```
case GATTC_USER_CHARACTERISTICS_DESCRIP_WRITE_RESULT:
{
    // errcode indicates whether the write operation is successful. 0 indicates
    // success.
    PRINTF("write descryptor over [%x]\n", errcode); // write over;
    break;
}
```

Modify MTU

To change the MTU, the following code can be executed:

```
gatt_client_MTU_exchange (0, _mtu);
```

The `_mtu` parameter is the local supported MTU. This operation will negotiate a MTU supported by both devices, and then return in the callback function:

```
case GATTC_USER_MTU_EXCHG:
{
    if (buf && (errcode == 0))
    {
        // The first two bytes of buf are the negotiated MTU.
        // If the first two bytes are 08 and 01, the MTU is 0x0108
        uint16_t mtu = little_endian_read_16(buf, 0);
        PRINTF("exchang mtu [len%d--used mtu=%d]:", len, mtu);
    }
    if (isClientOver)
    {
        PRINTF("exchang mtu over [%x]\n", errcode);
    }
    break;
}
```

Notification and Indication Data

The callback function will return the notification and Indication data sent by the peer, as follows:

```
case GATTC_USER_NOTIFICATION:
{
    /* After receiving the notification data, the first and second bytes in buf are
    the handle of the characteristic for sending data, and the following data
    is the notification data, its length is len-2 */
    uint16_t value_handle = little_endian_read_16(buf, 1);
    if (g_rx_char.handle == value_handle)
    {
        // This is the data sent by the uplink characteristic in the transparent
        // service
        PRINTF("notify data len= [%d]\n", len - 2);
    }
}
case GATTC_USER_INDICATION:
{
    /* After receiving the indication data, the first and second bytes in buf are
    the handle of the characteristic for sending data, and the following data
    is the indication data, its length is len-2 */
    uint16_t value_handle = little_endian_read_16(buf, 1);
    if (g_indicate_char.handle == value_handle)
    {
```

```
// This is the data sent by the characteristics for testing Indication
// in the transparent service
PRINTF("recv indication, data len= [%d]\n", len - 2);
}
}
```

Client Transparent Profile Implement Summary

Through the Client operations in the previous examples, the transparent profile can be easily implemented:

1. Find the transparent service by querying the primary service.
2. Query all characteristics and their descriptors contained in the transparent service.
3. Modify the Client Config descriptor of the uplink characteristic to enable it to send notification data.
4. Perform a data transparent transmission test. Send data to the peer by writing data downlink characteristics.
5. Perform a data transparent transmission test. The peer sends the uplink data by sending notifications and the notification data is received from the peer in the callback function.
6. Perform other operation tests as required.

Appendix

Common UUID References

GATT Services

```
#define GAP_SERVICE_UUID          0x1800 // Generic Access Profile
#define GATT_SERVICE_UUID        0x1801 // Generic Attribute Profile
```

GATT Declarations

```
#define GATT_PRIMARY_SERVICE_UUID 0x2800 // Primary Service
#define GATT_SECONDARY_SERVICE_UUID 0x2801 // Secondary Service
#define GATT_INCLUDE_SERVICE_UUID 0x2802 // Include
#define GATT_CHARACTERISTICS_UUID 0x2803 // Characteristic
```

GATT Descriptors

```
#define GATT_CHAR_EXT_PROPS_UUID 0x2900 // Characteristic Extended Properties
#define GATT_CHAR_USER_DESC_UUID 0x2901 // Characteristic User Description
#define GATT_CLIENT_CHAR_CFG_UUID 0x2902 // Client Characteristic Configuration
#define GATT_SERV_CHAR_CFG_UUID 0x2903 // Server Characteristic Configuration
#define GATT_CHAR_FORMAT_UUID 0x2904 // Characteristic Presentation Format
#define GATT_CHAR_AGG_FORMAT_UUID 0x2905 // Characteristic Aggregate Format
#define GATT_VALID_RANGE_UUID 0x2906 // Valid Range
#define GATT_EXT_REPORT_REF_UUID 0x2907 // External Report Reference Descriptor
#define GATT_REPORT_REF_UUID 0x2908 // Report Reference Descriptor
```

GATT Characteristics

```
#define DEVICE_NAME_UUID          0x2A00 // Device Name
#define APPEARANCE_UUID          0x2A01 // Appearance
#define PERI_PRIVACY_FLAG_UUID 0x2A02 // Peripheral Privacy Flag
#define RECONNECT_ADDR_UUID      0x2A03 // Reconnect Address
#define PERI_CONN_PARAM_UUID     0x2A04 // Peripheral Preferred Connection
// Parameters
#define SERVICE_CHANGED_UUID      0x2A05 // Service Changed
```

GATT Service UUID

```
#define IMMEDIATE_ALERT_SERV_UUID 0x1802 // Immediate Alert
#define LINK_LOSS_SERV_UUID        0x1803 // Link Loss
#define TX_PWR_LEVEL_SERV_UUID     0x1804 // TX Power
#define CURRENT_TIME_SERV_UUID     0x1805 // Current Time Service
#define REF_TIME_UPDATE_SERV_UUID  0x1806 // Reference Time Update Service
#define NEXT_DST_CHANGE_SERV_UUID  0x1807 // Next DST Change Service
#define GLUCOSE_SERV_UUID           0x1808 // Glucose
#define THERMOMETER_SERV_UUID       0x1809 // Health Thermometer
#define DEVINFO_SERV_UUID           0x180A // Device Information
#define NWA_SERV_UUID               0x180B // Network Availability
#define HEARTRATE_SERV_UUID         0x180D // Heart Rate
#define PHONE_ALERT_STS_SERV_UUID   0x180E // Phone Alert Status Service
#define BATT_SERV_UUID              0x180F // Battery Service
#define BLOODPRESSURE_SERV_UUID     0x1810 // Blood Pressure
#define ALERT_NOTIF_SERV_UUID       0x1811 // Alert Notification Service
#define HID_SERV_UUID               0x1812 // Human Interface Device
#define SCAN_PARAM_SERV_UUID        0x1813 // Scan Parameters
#define RSC_SERV_UUID               0x1814 // Running Speed and Cadence
#define CSC_SERV_UUID               0x1816 // Cycling Speed and Cadence
#define CYCPWR_SERV_UUID            0x1818 // Cycling Power
#define LOC_NAV_SERV_UUID            0x1819 // Location and Navigation
```

GATT Characteristic UUID

```
#define GAP_DEVICE_NAME_UUID        0x2a00
#define GAP_APPEARANCE_UUID         0x2a01
#define GAP_PERIPHERAL_PRIVACY_FLAG 0x2a02
#define GAP_RECONNECTION_ADDRESS_UUID 0x2a03
#define GAP_PERIPHERAL_PREFERRED_CONNECTION_PARAMETERS_UUID 0x2a04
#define GAP_SERVICE_CHANGED         0x2a05
#define ALERT_LEVEL_UUID             0x2A06 // Alert Level
#define TX_PWR_LEVEL_UUID            0x2A07 // TX Power Level
#define DATE_TIME_UUID               0x2A08 // Date Time
#define DAY_OF_WEEK_UUID             0x2A09 // Day of Week
#define DAY_DATE_TIME_UUID           0x2A0A // Day Date Time
#define EXACT_TIME_256_UUID           0x2A0C // Exact Time 256
#define DST_OFFSET_UUID              0x2A0D // DST Offset
#define TIME_ZONE_UUID               0x2A0E // Time Zone
#define LOCAL_TIME_INFO_UUID          0x2A0F // Local Time Information
#define TIME_WITH_DST_UUID            0x2A11 // Time with DST
#define TIME_ACCURACY_UUID            0x2A12 // Time Accuracy
#define TIME_SOURCE_UUID              0x2A13 // Time Source
#define REF_TIME_INFO_UUID            0x2A14 // Reference Time Information
#define TIME_UPDATE_CTRL_PT_UUID      0x2A16 // Time Update Control Point
#define TIME_UPDATE_STATE_UUID        0x2A17 // Time Update State
#define GLUCOSE_MEAS_UUID             0x2A18 // Glucose Measurement
#define BATT_LEVEL_UUID               0x2A19 // Battery Level
#define TEMP_MEAS_UUID                0x2A1C // Temperature Measurement
#define TEMP_TYPE_UUID                0x2A1D // Temperature Type
#define IMEDIATE_TEMP_UUID            0x2A1E // Intermediate Temperature
#define MEAS_INTERVAL_UUID            0x2A21 // Measurement Interval
#define BOOT_KEY_INPUT_UUID           0x2A22 // Boot Keyboard Input Report
#define SYSTEM_ID_UUID                0x2A23 // System ID
#define MODEL_NUMBER_UUID             0x2A24 // Model Number String
#define SERIAL_NUMBER_UUID            0x2A25 // Serial Number String
#define FIRMWARE_REV_UUID             0x2A26 // Firmware Revision String
#define HARDWARE_REV_UUID             0x2A27 // Hardware Revision String
#define SOFTWARE_REV_UUID             0x2A28 // Software Revision String
```

```
#define MANUFACTURER_NAME_UUID      0x2A29 // Manufacturer Name String
#define IEEE_11073_CERT_DATA_UUID   0x2A2A // IEEE 11073-20601 Regulatory
                                         // Certification Data List

#define CURRENT_TIME_UUID           0x2A2B // Current Time
#define SCAN_REFRESH_UUID           0x2A31 // Scan Refresh
#define BOOT_KEY_OUTPUT_UUID        0x2A32 // Boot Keyboard Output Report
#define BOOT_MOUSE_INPUT_UUID       0x2A33 // Boot Mouse Input Report
#define GLUCOSE_CONTEXT_UUID        0x2A34 // Glucose Measurement Context
#define BLOODPRESSURE_MEAS_UUID     0x2A35 // Blood Pressure Measurement
#define IMMEDIATE_CUFF_PRESSURE_UUID 0x2A36 // Intermediate Cuff Pressure
#define HEARTRATE_MEAS_UUID         0x2A37 // Heart Rate Measurement
#define BODY_SENSOR_LOC_UUID        0x2A38 // Body Sensor Location
#define HEARTRATE_CTRL_PT_UUID      0x2A39 // Heart Rate Control Point
#define NETWORK_AVAIL_UUID          0x2A3E // Network Availability
#define ALERT_STATUS_UUID           0x2A3F // Alert Status
#define RINGER_CTRL_PT_UUID         0x2A40 // Ringer Control Point
#define RINGER_SETTING_UUID         0x2A41 // Ringer Setting
#define ALERT_CAT_ID_BMASK_UUID     0x2A42 // Alert Category ID Bit Mask
#define ALERT_CAT_ID_UUID           0x2A43 // Alert Category ID
#define ALERT_NOTIF_CTRL_PT_UUID     0x2A44 // Alert Notification Control Point
#define UNREAD_ALERT_STATUS_UUID    0x2A45 // Unread Alert Status
#define NEW_ALERT_UUID              0x2A46 // New Alert
#define SUP_NEW_ALERT_CAT_UUID       0x2A47 // Supported New Alert Category
#define SUP_UNREAD_ALERT_CAT_UUID    0x2A48 // Supported Unread Alert Category
#define BLOODPRESSURE_FEATURE_UUID   0x2A49 // Blood Pressure Feature
#define HID_INFORMATION_UUID         0x2A4A // HID Information
#define REPORT_MAP_UUID             0x2A4B // Report Map
#define HID_CTRL_PT_UUID            0x2A4C // HID Control Point
#define REPORT_UUID                 0x2A4D // Report
#define PROTOCOL_MODE_UUID          0x2A4E // Protocol Mode
#define SCAN_INTERVAL_WINDOW_UUID    0x2A4F // Scan Interval Window
#define PNP_ID_UUID                 0x2A50 // PnP ID
#define GLUCOSE_FEATURE_UUID         0x2A51 // Glucose Feature
#define RECORD_CTRL_PT_UUID         0x2A52 // Record Access Control Point
#define RSC_MEAS_UUID               0x2A53 // RSC Measurement
#define RSC_FEATURE_UUID            0x2A54 // RSC Feature
#define SC_CTRL_PT_UUID             0x2A55 // SC Control Point
#define CSC_MEAS_UUID               0x2A5B // CSC Measurement
#define CSC_FEATURE_UUID            0x2A5C // CSC Feature
#define SENSOR_LOC_UUID             0x2A5D // Sensor Location
#define CYCPWR_MEAS_UUID            0x2A63 // Cycling Power Measurement
#define CYCPWR_VECTOR_UUID          0x2A64 // Cycling Power Vector
#define CYCPWR_FEATURE_UUID         0x2A65 // Cycling Power Feature
#define CYCPWR_CTRL_PT_UUID         0x2A66 // Cycling Power Control Point
#define LOC_SPEED_UUID              0x2A67 // Location and Speed
#define NAV_UUID                    0x2A68 // Navigation
#define POS_QUALITY_UUID            0x2A69 // Position Quality
#define LN_FEATURE_UUID             0x2A6A // LN Feature
#define LN_CTRL_PT_UUID             0x2A6B // LN Control Point
```

Appearance Attribute Definition

```
#define GAP_APPEARE_UNKNOWN         0x0000 // Unknown
#define GAP_APPEARE_GENERIC_PHONE   0x0040 // Generic Phone
#define GAP_APPEARE_GENERIC_COMPUTER 0x0080 // Generic Computer
#define GAP_APPEARE_GENERIC_WATCH   0x00C0 // Generic Watch
#define GAP_APPEARE_WATCH_SPORTS    0x00C1 // Watch: Sports Watch
#define GAP_APPEARE_GENERIC_CLOCK   0x0100 // Generic Clock
#define GAP_APPEARE_GENERIC_DISPLAY 0x0140 // Generic Display
```



```
#define GAP_APPEARE_GENERIC_RC                0x0180 // Generic Remote Control
#define GAP_APPEARE_GENERIC_EYE_GALSSES      0x01C0 // Generic Eye-glasses
#define GAP_APPEARE_GENERIC_TAG               0x0200 // Generic Tag
#define GAP_APPEARE_GENERIC_KEYRING          0x0240 // Generic Keyring
#define GAP_APPEARE_GENERIC_MEDIA_PLAYER     0x0280 // Generic Media Player
#define GAP_APPEARE_GENERIC_BARCODE_SCANNER  0x02C0 // Generic Barcode Scanner
#define GAP_APPEARE_GENERIC_THERMOMETER      0x0300 // Generic Thermometer
#define GAP_APPEARE_GENERIC_THERMO_EAR       0x0301 // Thermometer: Ear
#define GAP_APPEARE_GENERIC_HR_SENSOR        0x0340 // Generic Heart rate Sensor
#define GAP_APPEARE_GENERIC_HRS_BELT         0x0341 // Heart Rate Sensor: Heart Rate
                                                // Belt
#define GAP_APPEARE_GENERIC_BLOOD_PRESSURE   0x0380 // Generic Blood Pressure
#define GAP_APPEARE_GENERIC_BP_ARM           0x0381 // Blood Pressure: Arm
#define GAP_APPEARE_GENERIC_BP_WRIST        0x0382 // Blood Pressure: Wrist
#define GAP_APPEARE_GENERIC_HID              0x03C0 // Human Interface Device (HID)
#define GAP_APPEARE_HID_KEYBOARD             0x03C1 // HID Keyboard
#define GAP_APPEARE_HID_MOUSE                0x03C2 // HID Mouse
#define GAP_APPEARE_HID_JOYSTIC              0x03C3 // HID Joystick
#define GAP_APPEARE_HID_GAMEPAD              0x03C4 // HID Gamepad
#define GAP_APPEARE_HID_DIGITIZER_TYABLET    0x03C5 // HID Digitizer Tablet
#define GAP_APPEARE_HID_DIGITAL_CARDREADER   0x03C6 // HID Card Reader
#define GAP_APPEARE_HID_DIGITAL_PEN          0x03C7 // HID Digital Pen
#define GAP_APPEARE_HID_BARCODE_SCANNER      0x03C8 // HID Barcode Scanner
```

Copyright© 2025 by HOLTEK SEMICONDUCTOR INC. All Rights Reserved.

The information provided in this document has been produced with reasonable care and attention before publication, however, HOLTEK does not guarantee that the information is completely accurate. The information contained in this publication is provided for reference only and may be superseded by updates. HOLTEK disclaims any expressed, implied or statutory warranties, including but not limited to suitability for commercialization, satisfactory quality, specifications, characteristics, functions, fitness for a particular purpose, and non-infringement of any third-party's rights. HOLTEK disclaims all liability arising from the information and its application. In addition, HOLTEK does not recommend the use of HOLTEK's products where there is a risk of personal hazard due to malfunction or other reasons. HOLTEK hereby declares that it does not authorize the use of these products in life-saving, life-sustaining or safety critical components. Any use of HOLTEK's products in life-saving/sustaining or safety applications is entirely at the buyer's risk, and the buyer agrees to defend, indemnify and hold HOLTEK harmless from any damages, claims, suits, or expenses resulting from such use. The information provided in this document, including but not limited to the content, data, examples, materials, graphs, and trademarks, is the intellectual property of HOLTEK (and its licensors, where applicable) and is protected by copyright law and other intellectual property laws. No license, express or implied, to any intellectual property right, is granted by HOLTEK herein. HOLTEK reserves the right to revise the information described in the document at any time without prior notice. For the latest information, please contact us.