QN9020 API Programming Guide

Version 1.1

Table of Contents

1. Introduction

The purpose of this document is to give guide of the Quintic QN9020 Bluetooth Low Energy (BLE) API programming for software development. Quintic QN9020 solution offers a complete Software Development Kit (SDK) to develop various single-mode BLE applications. This file contains the documentation of all QN9020 BLE APIs in SDK, including QN9020 driver APIs, BLE protocol stack APIs, and application profile APIs. In addition, it demonstrates detailed driver and profile examples to help you get start on it.

2. QN9020 Driver

This chapter in the user manual describes Quintic QN9020 modules which conform to the Cortex Microcontroller Software Interface Standard (CMSIS). It also includes drivers for the following list of modules:

ADC Driver
Analog Driver
DMA Driver
Driver Configurations
GPIO Driver
I2C Driver
PWM Driver
RTC Driver
SPI Driver
Serial Flash Driver
Sleep Driver
System Controller Driver
Timer Driver
UART Driver
WDT Driver

The Driver contains C and assembly functions that have been ported and tested on the MDK toolchain.

2.1 ADC Driver

Detailed Description

QN9020 contains an up to 12 bits resolution successive approximation analog-to-digital converter (SAR A/D converter) with 12 input channels. It takes about 20 ADC clock cycles to convert one sample, and the maximum input clock to ADC is 1MHz. The A/D converter supports multi operation modes and can be started by 4 types of trigger sources.

The main features of ADC are listed as follow:

Maximum sample rate is 1MSPS
Support 8/10/12 bits resolution for one sample data
ADC input can be selected from 6 sources which includes 4 single-end input and 2 differential input
ADC conversion can be triggered by 4 sources: Software Start, Timer0/1 overflow and GPIO
Support selectable decimation rates, thereby corresponding improved effective resolutions

Support window compare, and generate corresponding interrupt
Support single conversion mode, continuous conversion mode
Support single scan conversion mode, continuous scan conversion mode
Support up to 1MHz/20 sampling rate
Support DMA
Support selectable reference voltage

Data Structure Documentation

struct adc_init_configuration

Data Fields:

enum ADC_WORK_CLK	work_clk	ADC work clock
enum ADC_REF	ref_vol	ADC reference voltage
enum ADC_RESOLUTION	resolution	ADC resolution
enum BUFF_IN_TYPE	buf_in_p	ADC input buffer P
enum BUFF_IN_TYPE	buf_in_n	ADC input buffer N
enum ADC_BUFF_GAIN	gain	ADC input buffer gain

struct adc_read_configuration

Data Fields:

enum ADC_WORK_MOD	mode	ADC work mode
enum ADC_TRIG_SRC	trig_src	ADC trigger source
enum ADC_CH	start_ch	ADC start channel
enum ADC_CH	end_ch	ADC end channel

Macro Definition Documentation

#define CFG_ADC_EXT_REF_VOL (3000)

External reference voltage: mV (CFG_ADC_EXT_REF_VOL = 2*EXT_REF1 or CFG_ADC_EXT_REF_VOL = EXT_REF2)

Enumeration Type Documentation

enum ADC_IN_MOD

ADC input mode.

Enumerator:

ADC_DIFF_WITH_BUF_DRV ADC differential input with buffer, input singal 0.2 =< VIN(V) <= VDD-0.2, ADC result [-2048, 2047] map to [-VREF, VREF).

ADC_DIFF_WITHOUT_BUF_DRV ADC differential input without buffer, input singal 0 =< VIN(V) <= VDD, and should have enough driving capability, ADC result [-2048, 2047] map to [-VREF, VREF).

ADC_SINGLE_WITH_BUF_DRV ADC single-ended input with buffer, input singal 0.2 =< VIN(V) <= 1.5*VREF <= VDD-0.2, ADC result [x, 2047] map to [0.2, 1.5*VREF).

ADC_SINGLE_WITHOUT_BUF_DRV ADC single-ended input without buffer, input singal 0 =< VIN(V) <= VREF <= VDD, and should have enough driving capability, ADC result [0, 2047] map to [0, VREF).

enum ADC_CH

ADC channel index.

Enumerator:

AIN0 Analog single channel 0, P3.0

AIN1 Analog single channel 1, P3.1

AIN2 Analog single channel 2, P0.6

AIN3 Analog single channel 3, P0.7

AIN01 Analog differential channel 0/1, P3.0/P3.1

AIN23 Analog differential channel 2/3, P0.6/P0.7

TEMP temperture sensor channel

BATT Battery detector channel

enum ADC_WORK_MOD

ADC work mode.

Enumerator:

SINGLE_MOD Single mode,

CONTINUE_MOD Continue mode, only need trigger once

SINGLE_SCAN_MOD Single scan mode

CONTINUE_SCAN_MOD Continue scan mode

enum ADC_REF

ADC reference voltage.

Enumerator:

ADC_INT_REF Internal reference, VREF = 1.0V

ADC_EXT_REF1 External reference1(with buffer and gain=2, input PIN is P0.7): VREF = 2*EXT_REF1 (0 < EXT_REF1 < (VDD-1.0)/2).

ADC_EXT_REF2 External reference2(without buffer, input PIN is P0.7): VERF = EXT_REF2 (0 < EXT_REF2 < VDD), EXT_REF2 should have driving capability.

enum ADC_TRIG_SRC

ADC Trigger source.

Enumerator:

ADC_TRIG_SOFT Triggered by software

ADC_TRIG_TOVF0 Triggered by timer0 overflow

ADC_TRIG_TOVF1 Triggered by timer1 overflow

ADC_TRIG_GPIO Triggered by GPIO

ADC_TRIG_CALIB Triggered by Calibration

enum ADC_GPIO_TRIG

ADC GPIO trigger PIN.

Enumerator:

ADC_GPIO06_TRIG Triggered by GPIO06
ADC_GPIO15_TRIG Triggered by GPIO15

enum ADC_RESOLUTION

ADC resolution.

Enumerator:

ADC_12BIT 12 bits resolution
ADC_10BIT 10 bits resolution
ADC_8BIT 8 bits resolution

enum ADC_CLK_SRC

ADC clock source.

Enumerator:

CLK_HIGH 32MHz or 16MHz, depends on system clock
CLK_LOW 32KHz

enum ADC_WORK_CLK

ADC working clock(ADC_SOURCE_CLK / (2<<ADC_DIV), ADC_SOURCE_CLK not from AHB)

Enumerator:

ADC_CLK_1000000 ADC work at 1MHz, when clock source is 16MHz
ADC_CLK_500000 ADC work at 500KHz, when clock source is 16MHz
ADC_CLK_250000 ADC work at 250KHz, when clock source is 16MHz
ADC_CLK_125000 ADC work at 125KHz, when clock source is 16MHz
ADC_CLK_62500 ADC work at 62.5KHz, when clock source is 16MHz
ADC_CLK_31250 ADC work at 31.25KHz, when clock source is 16MHz
ADC_CLK_15625 ADC work at 15.625KHz, when clock source is 16MHz
ADC_CLK32K_16000 ADC work at 16KHz, when clock source is 32KHz
ADC_CLK32K_8000 ADC work at 8KHz, when clock source is 32KHz
ADC_CLK32K_4000 ADC work at 4KHz, when clock source is 32KHz
ADC_CLK32K_2000 ADC work at 2KHz, when clock source is 32KHz
ADC_CLK32K_1000 ADC work at 1KHz, when clock source is 32KHz
ADC_CLK32K_500 ADC work at 500Hz, when clock source is 32KHz
ADC_CLK32K_250 ADC work at 250Hz, when clock source is 32KHz
ADC_CLK32K_125 ADC work at 125Hz, when clock source is 32KHz

enum WCMP_DATA

Window comparator data source.

Enumerator:

ADC_DATA ADC raw data
DECI_DATA Decimation data

enum DECIMATION_RATE

Decimation rate.

Enumerator:

DECI_RATE_64 Decimation rate: 64
DECI_RATE_256 Decimation rate: 256
DECI_RATE_1024 Decimation rate: 1024

enum BUFF_IN_TYPE

ADC buffer input type.

Enumerator:

ADC_BUFIN_VCM VCM
ADC_BUFIN_CHANNEL ADC channel
ADC_BUFIN_GND GND

enum ADC_BUFF_GAIN

ADC input buffer gain control.

Enumerator:

ADC_BUF_NEG_6DB -6dB
ADC_BUF_0DB 0dB
ADC_BUF_POS_6DB 6dB
ADC_BUF_POS_12DB 12dB
ADC_BUF_GAIN_BYPASS Bypass ADC input buffer gain stage
ADC_BUF_BYPASS Bypass ADC input buffer

Function Documentation

static void __adc_cofig (const adc_init_configuration * *S*)[static]

ADC configuration.

Parameters:

in	*S*	ADC initial configuration, contains work clock, reference voltage selection, resolution and input buffer setting

Description

This function is used to configure the ADC.

static void __adc_calibrate (const adc_init_configuration * *S*)[static]

ADC calibration.

Parameters:

in	*S*	ADC initial configuration, contains work clock, reference voltage selection, resolution and input buffer setting

Description

This function is used to get the ADC calibration result

static void __adc_offset_get (void)[static]

Get ADC offset for conversion result correction.

Description

This function is used to get ADC offset for conversion result correction, and should be called after ADC initialization and buffer gain settings.

void adc_clean_fifo (void)

Clean ADC FIFO.

Description

This function is used to clean ADC FIFO.

void adc_init (enum ADC_IN_MOD in_mod, enum ADC_WORK_CLK work_clk, enum ADC_REF ref_vol, enum ADC_RESOLUTION resolution)

Initialize ADC.

Parameters:

in	*in_mod*	ADC input mode
in	*work_clk*	ADC work_clk = (ADC_SOURCE_CLK / (2<<ADC_DIV)), ADC_DIV = 0 ~ 15, ADC_SOURCE_CLK is 32k or system clock(4 types, decided by CLK_MUX), the max work_clk = 1MHz.
in	*ref_vol*	ADC reference voltage
in	*resolution*	ADC resolution

Description

This function is used to set ADC input mode, work clock, reference voltage, resolution, and interrupt.

void adc_read (const adc_read_configuration * *S*, int16_t * *buf*, uint32_t *samples*, void(*)(void) *callback*)

Read ADC conversion result.

Parameters:

in	*S*	ADC read configuration, contains work mode, trigger source, start/end channel
in	*buf*	ADC result buffer
in	*samples*	Sample number
in	*callback*	callback after all the samples conversion finish

Description

This function is used to read ADC specified channel conversion result.

Note:

When use scaning mode, only can select first 6 channel (AIN0,AIN1,AIN2,AIN3,AIN01,AIN23)

void adc_buf_in_set (enum BUFF_IN_TYPE *buf_in_p*, enum BUFF_IN_TYPE *buf_in_n*)

Set ADC buffer input source.

Parameters:

in	*buf_in_p*	ADC Buffer input+
in	*buf_in_n*	ADC Buffer input-

Description

This function is used to set ADC buffer input source

void adc_buf_gain_set (enum ADC_BUFF_GAIN *gain*)

ADC buffer gain set.

Parameters:

in	*gain*	ADC buffer gain stage

Description

This function is used to set ADC buffer gain stage, and only available at the input mode with buffer driver.

void adc_compare_init (enum WCMP_DATA *data*, int16_t *high*, int16_t *low*, void(*)(void) *callback*)

Initialize ADC comparator.

Parameters:

in	*data*	data source of comparator
in	*high*	high level of compare window: higher than this level will generate interrupt
in	*low*	lwo level of compare window: lower than this level will generate interrupt
in	*callback*	callback after interrupt

Description

This function is used to initialize ADC window comparator.

void adc_decimation_enable (enum DECIMATION_RATE *rate*, uint32_t *able*)

Enable/Disable ADC decimation.

Parameters:

in	*rate*	decimation rate
in	*able*	mask of enable or disable

Description

This function is used to enable or disable ADC decimation

int16_t ADC_RESULT_mV (int16_t *adc_data*)

ADC result(mv)

Parameters:

in	*adc_data*	ADC data

Returns:

voltage value(mv)

Description

This function is used to calculate ADC voltage value

__STATIC_INLINE void adc_enable (uint32_t *able*)

Enable or disable adc.

Parameters:

in	*able*	MASK_ENABLE or MASK_DISABLE

Description

This function is used to enable or disable ADC module.

__STATIC_INLINE void adc_clock_on (void)

Enable ADC module clock.

Description

This function is used to enable ADC module clock

__STATIC_INLINE void adc_clock_off (void)

Disable ADC module clock.

Description

This function is used to disable ADC module clock

__STATIC_INLINE void adc_power_on (void)

Power on ADC.

Description

This function is used to power on ADC module

__STATIC_INLINE void adc_power_off (void)

Power off ADC.

Description

This function is used to power off ADC module

__STATIC_INLINE void adc_reset (void)

Reset ADC module.

Description

This function is used to reset ADC module

__STATIC_INLINE void adc_pin_enable (enum ADC_CH *ainx*, uint32_t *able*)

Enable or disable analog input pin .

Parameters:

in	*ainx*	Analog pin index
in	*able*	MASK_ENABLE or MASK_DISABLE

Description

This function is used to enable or disable analog input pin.

Variable Documentation

struct adc_env_tag adc_env[static]

ADC environment variable.

volatile uint8_t scan_ch_num[static]

ADC SCAN channel number.

2.2 Analog Driver

Detailed Description

QN9020 analog circuit contains: clock generator, two comparators, ADC, battery monitor, brown out detector, temperature sensor, RF, power and reset modules. Please refer to system controller driver for how to control clock generator, as well as power and reset modules. Also please refer to RF driver for how to set frequecny, and refer to ADC driver for how to use ADC, The other modules are described in this section as well. Their main features are listed as follow:

Two comparators with selectable reference voltage
Interrupt generate according to comparator result
Support brown out detection
Intergrated temperature sensor

Macro Definition Documentation

#define DEFAULT_TEMP_OFFSET (-200)

default temperature offset value

#define FAC_CAL_TEMP (25)

factory calibartion temperature

#define TEMPERATURE_X10(adc_data) ((int16_t)(((((adc_data) - TEMP_OFFSET) / 3.8) + FAC_CAL_TEMP) * 10))

temperature calculated by ADC result

Enumeration Type Documentation

enum ACMP_CH

Analog comparator channel.

Enumerator:

ACMP0 Analog comparator channel 0

ACMP1 Analog comparator channel 1

enum ACMP_REF

Analog comparator reference voltage.

Enumerator:

EXT_REF Set reference valtage to external reference voltage

VDD_1 Set reference valtage to 1/16 VDD

VDD_2 Set reference valtage to 2/16 VDD

VDD_3 Set reference valtage to 3/16 VDD

VDD_4 Set reference valtage to 4/16 VDD

VDD_5 Set reference valtage to 5/16 VDD

VDD_6 Set reference valtage to 6/16 VDD

VDD_7 Set reference valtage to 7/16 VDD

VDD_8 Set reference valtage to 8/16 VDD

VDD_9 Set reference valtage to 9/16 VDD

VDD_10 Set reference valtage to 10/16 VDD

VDD_11 Set reference valtage to 11/16 VDD

VDD_12 Set reference valtage to 12/16 VDD

VDD_13 Set reference valtage to 13/16 VDD

VDD_14 Set reference valtage to 14/16 VDD

VDD_15 Set reference valtage to 15/16 VDD

enum ACMP_INT_COND

Analog comparator interrupt condition.

Enumerator:

ACMPO_1_GEN_INT When ACMP output is 1, generate interrupt

ACMPO_0_GEN_INT When ACMP output is 0, generate interrupt

enum ACMP_HYST_STATUS

Analog comparator Hysteresis Status.

Enumerator:

HYST_DISABLE Disable Hysteresis

HYST_ENABLE Enable Hysteresis

enum ACMP_PIN

Analog comparator Pin Select.

Enumerator:

ACMP0_PIN_P Analog comparator Pin Select, P3.0

ACMP0_PIN_N Analog comparator Pin Select, P3.1

ACMP1_PIN_P Analog comparator Pin Select, P0.6

ACMP1_PIN_N Analog comparator Pin Select, P0.7

Function Documentation

void ACMP0_IRQHandler (void)

ACMP0 interrupt handler.

void ACMP1_IRQHandler (void)

ACMP1 interrupt handler.

void acmp_init (enum ACMP_CH *acmpch*, enum ACMP_REF *acmpref*, enum ACMP_INT_COND *acmpint*, enum ACMP_HYST_STATUS *acmphyst*, void(*)(void) *callback*)

Initialize and enable ACMP.

Parameters:

in	*acmpch*	ACMP0 or ACMP1
in	*acmpref*	ACMP voltage: external or internal VDD (ref pin: ACMPx_N) This parameter can be one of the following value: ● EXT_REF External reference. ● VDD_x Where x can be (1..15) to select internal reference voltage
in	*acmpint*	ACMP interrutp condition: acmp output 1 or 0 generate interrupt This parameter can be one of the following value: ● ACMP0_0_GEN_INT When ACMP output is 0, generate interrupt ● ACMP0_1_GEN_INT When ACMP output is 1, generate interrupt
in	*acmphyst*	ACMP Hysteresis status set This parameter can be one of the following value: ● HYST_DISABLE Disable Hysteresis ● HYST_ENABLE Enable Hysteresis
in	*callback*	Callback in interrupt handl

Description

This function is used to initialize specified analog comparator, and to register callback function.

void acmp_enable (enum ACMP_CH *acmpch*, enum ACMP_INT_COND *acmpint*, uint32_t *able*)

Enable ACMP with interrupt condition.

Parameters:

in	*acmpch*	ACMP0 or ACMP0
in	*acmpint*	ACMP interrutp condition: acmp output 1 or 0 generate interrupt This parameter can be one of the following value: ● ACMP0_0_GEN_INT When ACMP output is 1, generate interrupt ● ACMP0_1_GEN_INT When ACMP output is 0, generate interrupt
in	*able*	MASK_ENABLE or MASK_DISABLE

Description

This function is used to enable or disable specified analog comparator with interrupt condition. If Comparators aren't used during sleep, please set ACMP0 and ACMP1 to disabled for lower sleep leakage current.

void battery_monitor_enable (uint32_t *able*)

Enable/Disable battery monitor.

Parameters:

in	*able*	MASK_ENABLE or MASK_DISABLE

Description

This function is used to enable or disable battery monitor.

void brown_out_enable (uint32_t *able*)

Enable/Disable brown out detector.

Parameters:

in	*able*	MASK_ENABLE or MASK_DISABLE

Description

This function is used to enable or disable brown out detector.

void temp_sensor_enable (uint32_t *able*)

Enable/Disable temperature sensor.

Parameters:

in	*able*	MASK_ENABLE or MASK_DISABLE

Description

This function is used to enable or disable temperature sensor.

bool acmp_sleep_allowed (void)

Check analog comparator sleep is allowed or not.

Returns:

TRUE or FALSE

Description

This function is used to check the analog comparator sleep is allowed or not, pin ACMPx_O should be configured before this function is called.

__STATIC_INLINE void acmp_pin_enable (enum ACMP_PIN *acmp_pin*, uint32_t *able*)

Enable or disable analog input pin .

Parameters:

in	*acmp_pin*	ACMP pin index This parameter can be one of the following value: ACMP0_PIN_P P3.0 selected as ACMP0 pin P ACMP0_PIN_N P3.1 selected as ACMP0 pin N ACMP1_PIN_P P0.6 selected as ACMP1 pin P ACMP1_PIN_N P0.7 selected as ACMP1 pin N
in	*able*	MASK_ENABLE or MASK_DISABLE

Description

This function is used to enable or disable analog input pin.

Variable Documentation

struct acmp_env_tag acmp0_env[static]

Analog comparator0 environment variable.

struct acmp_env_tag acmp1_env[static]

Analog comparator1 environment variable.

2.3 DMA Driver

Detailed Description

QN9020 contains a single channel DMA controller, which supports 4 types transfer modes. Its features are listed as follow:

- Support fixed and increment address transfer
- Support 4 types transfer modes:
  - Memory to memory: support word, half word, byte aligned address
  - Peripheral to memory: support word, half word, byte aligned address
  - Memory to Peripheral: support word, half word, byte aligned address
  - Peripheral to Peripheral: only support word aligned address
- Programmable source address and destination address
- Support undefined length transfer
- Maximum fixed transfer length up to 2047 bytes
- There isn't arbitration among several DMA requests, only one DMA request is selected
- Support mask or unmask DMA request of peripheral
- Data FIFO width is 32bits with depth one
- DMA can be aborted immediately when in a transfer process by configuring DMA ABORT register. At the same time, DMA done interrupt will be generated
- A DMA done interrupt is generated after DMA done
- A DMA error interrupt is generated when AHB returns an error response

Macro Definition Documentation

#define DMA_UNDEFINE_LENGTH_EN FALSE

Enable undefined length transfers.

#define DMA_MASK_ALL_INT_EN (DMA_MASK_DONE_IE|DMA_MASK_ERROR_IE|DMA_MASK_INT_EN)

Mask of all DMA interrupt enable.

Enumeration Type Documentation

enum DMA_TRANS_MODE

DMA transfer mode

Enumerator:

DMA_TRANS_BYTE Set DMA transfer mode as byte transfer

DMA_TRANS_HALF_WORD Set DMA transfer mode as half word transfer

DMA_TRANS_WORD Set DMA transfer mode as word transfer

enum DMA_PERIPHERAL_TX

DMA tx peripheral index

Enumerator:

DMA_UART0_TX Set DMA TX peripheral to UART0 TX

DMA_UART1_TX Set DMA TX peripheral to UART1 TX

DMA_SPI0_TX Set DMA TX peripheral to SPI0 TX

DMA_SPI1_TX Set DMA TX peripheral to SPI1 TX

DMA_PROP_TX Set DMA TX peripheral to Proprietary TX

DMA_TX_MAX DMA TX peripheral Total bumber

enum DMA_PERIPHERAL_RX

DMA rx peripheral index

Enumerator:

DMA_UART0_RX Set DMA RX peripheral to UART0 RX

DMA_UART1_RX Set DMA RX peripheral to UART1 RX

DMA_SPI0_RX Set DMA RX peripheral to SPI0 RX

DMA_SPI1_RX Set DMA RX peripheral to SPI1 RX

DMA_PROP_RX Set DMA RX peripheral to Proprietary RX

DMA_ADC Set DMA RX peripheral to ADC

DMA_RX_MAX DMA RX peripheral Total bumber

enum DMA_STATE

DMA status.

Enumerator:

DMA_BUSY DMA busy

DMA_FREE DMA free

Function Documentation

void DMA_IRQHandler (void)

DMA interrupt handler.

void dma_init (void)

Initialize DMA controller.

Description

This function is used to clear callback pointer and enable DMA NVIC IRQ.

int dma_check_status (void)

Check DMA status.

Returns:

DMA status

Description

This function is used to check DMA status.

void dma_abort (void)

DMA abort.

Description

This function is used to abort current DMA transfer, and usually used in undefined transfer length mode.

void dma_memory_copy (uint32_t *src_addr*, uint32_t *dst_addr*, uint32_t *size*, void(*)(void) *callback*)

DMA memory copy.

Parameters:

in	*src_addr*	source start address
in	*dst_addr*	destination start address

in	*size*	size of transfer
in	*callback*	callback after transfer

Description

This function is used to transfer data from memory to memory by DMA.

void dma_tx (enum DMA_TRANS_MODE *mode*, uint32_t *src_addr*, enum DMA_PERIPHERAL_TX *dst_index*, uint32_t *size*, void(*)(void) *tx_callback*)

DMA form memory to fix.

Parameters:

in	*mode*	transfer mode: byte, half word, word
in	*src_addr*	source address
in	*dst_index*	destination peripheral index
in	*size*	size of transfer
in	*tx_callback*	callback after transfer

Description

This function is used to transfer data from memory to peripheral by DMA.

void dma_rx (enum DMA_TRANS_MODE *mode*, enum DMA_PERIPHERAL_RX *src_index*, uint32_t *dst_addr*, uint32_t *size*, void(*)(void) *rx_callback*)

DMA form fix to memory.

Parameters:

in	*mode*	transfer mode: byte, half word, word
in	*src_index*	source peripheral index
in	*dst_addr*	destination address
in	*size*	size of transfer, the max size is 0x7FF
in	*rx_callback*	callback after transfer

Description

This function is used to transfer data from peripheral to memory by DMA.

void dma_transfer (enum DMA_PERIPHERAL_RX *src_index*, enum DMA_PERIPHERAL_TX *dst_index*, uint32_t *size*, void(*)(void) *trans_callback*)

DMA form peripheral to peripheral.

Parameters:

in	*src_index*	source peripheral index
in	*dst_index*	destination peripheral index
in	*size*	size of transfer
in	*trans_callback*	callback after transfer finish

Description

This function is used to transfer data from peripheral to peripheral by DMA.

__STATIC_INLINE void dma_clock_on (void)

Enable DMA module clock.

Description

This function is used to enable DMA module clock

__STATIC_INLINE void dma_clock_off (void)

Disable DMA module clock.

Description

This function is used to disable DMA module clock

__STATIC_INLINE void dma_reset (void)

Reset DMA module.

Description

This function is used to reset DMA module

Variable Documentation

struct dma_env_tag dma_env[static]

Variable used to store DMA environment.

const uint32_t peripheral_dst[DMA_TX_MAX][static]

```
Initial value:=
{
    QN_UART0_BASE+0x00,
    QN_UART1_BASE+0x00,
    QN_SPI0_BASE+0x10,
    QN_SPI1_BASE+0x10,
    QN_PROP_BASE+0x00
}
```

DMA TX peripheral address.

const uint32_t peripheral_src[DMA_RX_MAX][static]

```
Initial value:=
{
    QN_UART0_BASE+0x04,
    QN_UART1_BASE+0x04,
    QN_SPI0_BASE+0x14,
    QN_SPI1_BASE+0x14,
    QN_PROP_BASE+0x04,
    QN_ADC_BASE+0x10
}
```

DMA RX peripheral address.

2.4 GPIO Driver

Detailed Description

QN9020 has up to 31 General Purpose I/O pins which can be shared with other function pins, depending on the pin mux configuration. The main features of GPIO are listed as follow:

Each one of the GPIO pins is independent and has the corresponding register bits to control the pin function mode and data.

The type of each I/O pins can be independently software configured as input, output, open-drain or pull-up mode.

Macro Definition Documentation

#define GPIO_P0

Value:(uint32_t)(GPIO_P00 | GPIO_P01 | GPIO_P02 | GPIO_P03 | \
GPIO_P04 | GPIO_P05 | GPIO_P06 | GPIO_P07)

P00 - P07

#define GPIO_P1

Value:(uint32_t)(GPIO_P10 | GPIO_P11 | GPIO_P12 | GPIO_P13 | \
GPIO_P14 | GPIO_P15 | GPIO_P16 | GPIO_P17)

P10 - P17

#define GPIO_P2

Value:(uint32_t)(GPIO_P20 | GPIO_P21 | GPIO_P22 | GPIO_P23 | \
GPIO_P24 | GPIO_P25 | GPIO_P26 | GPIO_P27)

P20 - P17

#define GPIO_P3

Value:(uint32_t)(GPIO_P30 | GPIO_P31 | GPIO_P32 | GPIO_P33 | \
GPIO_P34 | GPIO_P35 | GPIO_P36)

P30 - P36

#define GPIO_PIN_ALL (uint32_t)(GPIO_P0 | GPIO_P1 | GPIO_P2 | GPIO_P3)

All Pins

Typedef Documentation

typedef void(* gpio_callback_t)(enum gpio_pin pin)

Callback function pointer type for level detection.

Enumeration Type Documentation

enum gpio_pin

Enumeration of GPIO pins.

Enumerator:

GPIO_P00	PIN0.0
GPIO_P01	PIN0.1
GPIO_P02	PIN0.2
GPIO_P03	PIN0.3
GPIO_P04	PIN0.4
GPIO_P05	PIN0.5
GPIO_P06	PIN0.6
GPIO_P07	PIN0.7
GPIO_P10	PIN1.0
GPIO_P11	PIN1.1
GPIO_P12	PIN1.2
GPIO_P13	PIN1.3
GPIO_P14	PIN1.4
GPIO_P15	PIN1.5
GPIO_P16	PIN1.6
GPIO_P17	PIN1.7
GPIO_P20	PIN2.0
GPIO_P21	PIN2.1
GPIO_P22	PIN2.2

GPIO_P23 PIN2.3
GPIO_P24 PIN2.4
GPIO_P25 PIN2.5
GPIO_P26 PIN2.6
GPIO_P27 PIN2.7
GPIO_P30 PIN3.0
GPIO_P31 PIN3.1
GPIO_P32 PIN3.2
GPIO_P33 PIN3.3
GPIO_P34 PIN3.4
GPIO_P35 PIN3.5
GPIO_P36 PIN3.6

enum gpio_level

GPIO states (low/high)

Enumerator:

GPIO_LOW Set GPIO to low level
GPIO_HIGH Set GPIO to high level

enum gpio_direction

GPIO direction (input/output)

Enumerator:

GPIO_INPUT Set GPIO direction to input
GPIO_OUTPUT Set GPIO direction to output

enum gpio_pull

GPIO pull states (low/high)

Enumerator:

GPIO_HIGH_Z Set GPIO as high impedance mode
GPIO_PULL_DOWN Set GPIO as pull-down mode
GPIO_PULL_UP Set GPIO as pull-up mode
GPIO_PULL_RSVD Reserved

enum gpio_int_trig_type

GPIO interrupt triger type (falling edge/rising edge/low level/high level)

Enumerator:

GPIO_INT_FALLING_EDGE Set GPIO interrupt enabled by falling edge
GPIO_INT_RISING_EDGE Set GPIO interrupt enabled by rising edge
GPIO_INT_LOW_LEVEL Set GPIO interrupt enabled by low level
GPIO_INT_HIGH_LEVEL Set GPIO interrupt enabled by high level

enum gpio_wakeup_type

GPIO wakeup type.

Enumerator:

GPIO_WKUP_BY_HIGH Set GPIO wakeup by high level
GPIO_WKUP_BY_LOW Set GPIO wakeup by low level
GPIO_WKUP_BY_CHANGE Set GPIO wakeup by level change

Function Documentation

void GPIO_IRQHandler (void)

Handles GPIO interrupt, polling and process.

Description

Get interrupt pin by polling interrupt status register, and then execute the callback function if it was enabled.

void gpio_init (gpio_callback_t *p_callback*)

Initialize and configure the GPIO.

Parameters:

in	*p_callback*	Callback function pointer, which is called in IRQHandler.

Description

This function is used to initialize callback function pointer and enable GPIO NVIC IRQ.

enum gpio_level gpio_read_pin (enum gpio_pin *pin*)

Read GPIO pin level.

Parameters:

in	*pin*	Specify pin of GPIO: GPIO_P00~GPIO_P07 GPIO_P10~GPIO_P17 GPIO_P20~GPIO_P27 GPIO_P30~GPIO_P36

Returns:

The level of specified pin value: GPIO_LOW / GPIO_HIGH

Description

This function is used to get a specified GPIO pin's level..

void gpio_write_pin (enum gpio_pin *pin*, enum gpio_level *level*)

Write on GPIO pin.

Parameters:

in	*pin*	Specify pin of GPIO: GPIO_P00~GPIO_P07 GPIO_P10~GPIO_P17 GPIO_P20~GPIO_P27 GPIO_P30~GPIO_P36
in	*level*	Level: GPIO_LOW or GPIO_HIGH

Description

This function is used to set level high(1) or low(0) to a specified GPIO pin.

void gpio_set_direction (enum gpio_pin *pin*, enum gpio_direction *direction*)

Set direction (input or output) of a set of GPIO pins.

Parameters:

in	*pin*	Specify pin of GPIO: GPIO_P00~GPIO_P07 GPIO_P10~GPIO_P17 GPIO_P20~GPIO_P27 GPIO_P30~GPIO_P36
in	*direction*	Value: GPIO_INPUT / GPIO_OUTPUT

Description

It writes on direction register without impacting unselected GPIO pins.

uint32_t gpio_read_pin_field (uint32_t *pin_mask*)

Read a set of GPIO pins.

Parameters:

in	*pin_mask*	Pin mask of GPIO specify which pins to read

Returns:

Masked GPIO DATA register value

Description

It reads from input register without unselected GPIO pins value.

void gpio_write_pin_field (uint32_t *pin_mask*, uint32_t *level_value*)

Write a set of GPIO pins.

Parameters:

in	*pin_mask*	Pin mask of GPIO specify which pins to set
in	*level_value*	Mask bit value to set: 1:high level; 0:low level.

Description

It writes on output register without impacting unselected GPIO pins.

void gpio_set_direction_field (uint32_t *pin_mask*, uint32_t *direction_value*)

Set direction (input or output) of a set of GPIO pins.

Parameters:

in	*pin_mask*	Pin mask of GPIO specify which pins to set
in	*direction_value*	Value: GPIO_INPUT / GPIO_OUTPUT

Description

It writes on direction register without impacting unselected GPIO pins.

void gpio_toggle_pin (enum gpio_pin *pin*)

Toggle a GPIO pin.

Parameters:

in	*pin*	Specify pin of GPIO: GPIO_P00~GPIO_P07 GPIO_P10~GPIO_P17 GPIO_P20~GPIO_P27 GPIO_P30~GPIO_P36

Description

This function is used to set a specified GPIO pin to the opposite level that is currently appied..

void gpio_set_interrupt (enum gpio_pin *pin*, enum gpio_int_trig_type *trig_type*)

Set interrupt edge/level, sinlge/double, polarity.

Parameters:

in	*pin*	Specify pin of GPIO: GPIO_P00~GPIO_P07 GPIO_P10~GPIO_P17 GPIO_P20~GPIO_P27 GPIO_P30~GPIO_P36
in	*trig_type*	4 types: high/low level, rising/falling edge

Description

This function is used to configure a specified GPIO pin's interrupt.

void gpio_enable_interrupt (enum gpio_pin *pin*)

Enable interrupt on GPIO pin.

Parameters:

in	*pin*	Specify pin of GPIO: GPIO_P00~GPIO_P07 GPIO_P10~GPIO_P17 GPIO_P20~GPIO_P27 GPIO_P30~GPIO_P36

Description

This function is used to enable a specified GPIO pin's interrupt.

void gpio_disable_interrupt (enum gpio_pin *pin*)

Disable interrupt on GPIO pin.

Parameters:

in	*pin*	Specify pin of GPIO: GPIO_P00~GPIO_P07 GPIO_P10~GPIO_P17 GPIO_P20~GPIO_P27 GPIO_P30~GPIO_P36

Description

This function is used to disable a specified GPIO pin's interrupt.

void gpio_pull_set (enum gpio_pin *pin*, enum gpio_pull *pull_state*)

Set GPIO pin to specified mode.

Parameters:

in	*pin*	Specify pin of GPIO: GPIO_P00~GPIO_P07 GPIO_P10~GPIO_P17 GPIO_P20~GPIO_P27 GPIO_P30~GPIO_P36
in	*pull_state*	Pin mode: 00 : High-Z, 01 : Pull-down, 10 : Pull-up, 11 : Reserved

Description

This function is used to set a specified pin mode to a specified GPIO pin.

void gpio_wakeup_config (enum gpio_pin *pin*, enum gpio_wakeup_type *type*)

configure GPIO wakeup

Parameters:

in	*pin*	Wakeup pin: P0, P1
in	*type*	Wakeup type: high, low, change

Description

This function is used to configure GPIO wakeup pin.

bool gpio_sleep_allowed (void)

Check gpio sleep is allowed or not.

Returns:

TRUE or FALSE

Description

This function is used to check the gpio sleep is allowed or not.

__STATIC_INLINE void gpio_clock_on (void)

Enable GPIO module clock.

Description

This function is used to enable GPIO module clock

__STATIC_INLINE void gpio_clock_off (void)

Disable GPIO module clock.

Description

This function is used to disable GPIO module clock

__STATIC_INLINE void gpio_reset (void)

Reset GPIO module.

Description

This function is used to reset GPIO module

Variable Documentation

struct gpio_env_tag gpio_env = {NULL}[static]

GPIO environment variable.

2.5 I2C Driver

Detailed Description

I2C is a bi-directional serial bus with two wires that provides a simple and efficient method of data exchange between devices. The I2C standard is a true multi-master bus including collision detection and arbitration that prevents data corruption if two or more masters attempt to control the bus simultaneously.

For QN9020, I2C device could act as master or slave and I2C driver can help user to use I2C functions easily. The main features of I2C are listed as follow:

Both I2C master and slave control are supported
Master baud rate is configurable.
Supports up to 400Kbps baud rate.
Master & slave support both 8 bit and 10 bit address mode.
Master supports SCL synchronization, and bus arbitration.
Slave supports SCL stretching.
8 bit shift register for transform.

Macro Definition Documentation

#define QN9020_I2C_ADDR 0x1A

Define QN9020 I2C slave address.

#define I2C_SCL_RATIO(x) (((__APB_CLK/(20 * (x))) - 1) << I2C_POS_SCL_RATIO)

Define I2C ratio algorithm: I2C CLK(x) should less than or equal to __APB_CLK/20.

#define I2C_SLAVE_ADDR(x) ((x) << I2C_POS_SLAVE_ADDR)

Set QN9020 I2C slave address.

#define I2C_MASK_ALL_INT 0x0000003F /* 5 - 0 */

Mask of all I2C interrupt.

#define I2C_MAX_TIMEOUT 0x0000FFFF

Define I2C timeout time.

#define I2C_MASTER 0

Define I2C master mode.

#define I2C_SLAVE 1

Define I2C slave mode.

Enumeration Type Documentation

enum I2C_BUS_STATE

I2C bus state.

Enumerator:

I2C_BUS_FREE I2C bus free

I2C_BUS_BUSY I2C bus busy

enum I2C_OP_FSM

I2C operate status.

Enumerator:

I2C_OP_IDLE I2C idle

I2C_OP_WRDATA I2C write data

I2C_OP_SETADDR I2C set address

I2C_OP_RDDATA I2C read data

I2C_OP_ABORT I2C abort

I2C_OP_FINISH I2C operate finish

enum I2C_ERR_CODE

I2C error code.

Enumerator:

I2C_NO_ERROR I2C no error

I2C_CONFLICT I2C conflict

I2C_NO_ACK I2C no ack

I2C_TIMEOUT I2C timeout

Function Documentation

void I2C_IRQHandler (void)

I2C interrupt handler, deal with master mode only.

void i2c_init (uint32_t *speed*, uint8_t * *buffer*, uint16_t *size*)

Initialize the I2C controller.

Parameters:

in	*speed*	SCL 1K: I2C_SCL_RATIO(1000)
in	*buffer*	i2c buffer (point to a gobal memory)
in	*size*	i2c buffer len, = address size + data size

Description

This function is used to initialize I2C in master mode. SCL speed is up to 400KHz. The function is also used to enable I2c interrupt, and enable NVIC I2C IRQ.

enum I2C_BUS_STATE i2c_bus_check (void)

Check I2C bus is busy or free.

Returns:

Busy or free

static enum I2C_ERR_CODE i2c_read (uint8_t *saddr*)[static]

Start a data reception.

Parameters:

in	*saddr*	slave device address (7bits, without R/W bit)

Returns:

Error code

Description

This function is used to complete an I2C read transaction from start to stop. All the intermittent steps are handled in the interrupt handler while the interrupt is enabled. Before this function is called, the read length, write length, I2C master buffer, and I2C state need to be filled. Please refer to I2C_BYTE_READ(). As soon as the end of the data transfer is detected, the callback function is called.

static enum I2C_ERR_CODE i2c_write (uint8_t *saddr*)[static]

Start a data transmission.

Parameters:

in	*saddr*	slave device address(7bits, without R/W bit)

Returns:

Error code

Description

This function is used to complete an I2C write transaction from start to stop. All the intermittent steps are handled in the interrupt handler while the interrupt is enabled. Before this function is called, the read length, write length, I2C master buffer, and I2C state need to be filled. Please refer to I2C_BYTE_WRITE(). As soon as the end of the data transfer is detected, the callback function is called.

uint8_t I2C_BYTE_READ (uint8_t *saddr*, uint8_t *reg_addr*)

Read a byte data form i2c device.

Parameters:

in	*saddr*	slave device address(7bits, without R/W bit)
in	*reg_addr*	device register address

Returns:

reg_data read from i2c bus

Description

Read a byte data from slave device, the data address is 8 bits. If I2C device not need to specify a data address, the input param reg_addr should be set to 0, and i2c_env.i2cTxCount also should be set to 0.

uint8_t I2C_BYTE_READ2 (uint8_t *saddr*, uint16_t *reg_addr*)

Read a byte data form i2c device.

Parameters:

in	*saddr*	slave device address(7bits, without R/W bit)
in	*reg_addr*	device register address

Returns:

reg_data read from i2c bus

Description

Read a byte data from slave device, the data address is 16 bits

void I2C_nBYTE_READ (uint8_t *saddr*, uint8_t *reg_addr*, uint8_t * *buffer*, uint16_t *len*)

Read n byte data form i2c device.

Parameters:

in	*saddr*	slave device address(7bits, without R/W bit)
in	*reg_addr*	device register address
in	*buffer*	Pointer to read data buffer
in	*len*	read data length

Description

Read n byte data from slave device, read start address is 8 bits and the data will be stored in buffer, n is the specified length

void I2C_nBYTE_READ2 (uint8_t *saddr*, uint16_t *reg_addr*, uint8_t * *buffer*, uint16_t *len*)

Read n byte data form i2c device.

Parameters:

in	*saddr*	slave device address(7bits, without R/W bit)
in	*reg_addr*	device register address
in	*buffer*	Pointer to read data buffer
in	*len*	read data length

Description

Read n byte data from slave device, read start address is 16 bits and the data will be stored in buffer, n is the specified length

void I2C_BYTE_WRITE (uint8_t *saddr*, uint8_t *reg_addr*, uint8_t *reg_data*)

Write a byte data to i2c device *.

Parameters:

in	*saddr*	slave device address(7bits, without R/W bit)
in	*reg_addr*	device register address
in	*reg_data*	byte data

Description

Write a byte data to a 8 bits address of slave device

void I2C_BYTE_WRITE2 (uint8_t *saddr*, uint16_t *reg_addr*, uint8_t *reg_data*)

Write a byte data to i2c device *.

Parameters:

in	*saddr*	slave device address(7bits, without R/W bit)
in	*reg_addr*	device register address
in	*reg_data*	byte data

Description

Write a byte data to a 16 bits address of slave device

void I2C_nBYTE_WRITE (uint8_t *saddr*, uint8_t *reg_addr*, uint8_t * *buffer*, uint16_t *len*)

Write n byte data to i2c device *.

Parameters:

in	*saddr*	slave device address(7bits, without R/W bit)
in	*reg_addr*	device register address
in	*buffer*	pointer to write data
in	*len*	write data length

Description

Write n byte data to slave device. The write starting address is 8 bits. The data is from the buffer and n is a specified length

void I2C_nBYTE_WRITE2 (uint8_t *saddr*, uint16_t *reg_addr*, uint8_t * *buffer*, uint16_t *len*)

Write n byte data to i2c device *.

Parameters:

in	*saddr*	slave device address(7bits, without R/W bit)
in	*reg_addr*	device register address
in	*buffer*	pointer to write data
in	*len*	write data length

Description

Write n byte data to slave device. The write starting address is 16 bits. The data is from the buffer and n is a specified length

__STATIC_INLINE void i2c_reset (void)

Reset I2C module.

Description

This function is used to reset I2C module

Variable Documentation

struct i2c_env_tag i2c_env[static]

I2C environment variable.

2.6 PWM Driver

Detailed Description

QN9020 PWM module provides two channels with programmable period and duty cycle. The main features of PWM are listed as follow:

Two 8-bit auto-reload count down counter
Programmable 10-bit prescaler for both channels
Predictable PWM initial output state
Buffered compare register and polarity register to ensure correct PWM output
Programmable overflow interrupt generation

Macro Definition Documentation

#define PWM_DIV(n) ((n) + 1)

Set PWM divider.

#define PWM_CLK(x, n) (TIMER_CLK(x) / (PWM_DIV(n)))

Set PWM clock.

#define PWM_PSCAL_DIV 63

Set prescaler.

#define PWM_COUNT_S(s, pscl_div) ((s) * PWM_CLK(TIMER_DIV, pscl_div))

Set period&compare count value (periodInS * (PWM_CLK))

#define PWM_COUNT_MS(ms, pscl_div) ((ms) * PWM_CLK(TIMER_DIV, pscl_div) / 1000)

Set period&compare count value (periodInMs * (PWM_CLK / 1000))

#define PWM_COUNT_US(us, pscl_div) ((us) * PWM_CLK(TIMER_DIV, pscl_div) / 1000000)

Set period&compare count value (periodInUs * (PWM_CLK / 1000000))

Enumeration Type Documentation

enum PWM_CH

PWM channel.

Enumerator:

PWM_CH0 PWM channel 0
PWM_CH1 PWM channel 1

Function Documentation

void pwm_init (enum PWM_CH *pwmch*)

Initialize the PWM.

Parameters:

in	*pwmch*	PWM_CH0, PWM_CH1

Description

This function is used to initialize the specified PWM channel.

uint8_t pwm_config (enum PWM_CH *pwmch*, uint16_t *pscal*, uint8_t *periodcount*, uint8_t *pulsecount*)

Config the PWM.

Parameters:

in	*pwmch*	PWM_CH0, PWM_CH1
in	*pscal*	PWM prescaler value: 0x0 ~ 0x3FF
in	*periodcount*	period count: 0x0 ~ 0xFF
in	*pulsecount*	pulse count: 0x0 ~ 0xFF, pulsecount should less than periodcount

Returns:

success(1) or failed(0)

Description

This function is used to config the specified PWM channel. It contains configuraion of pre-scaler, period, and pulse width.

e.g: pwm_config(PWM_CH0, PWM_PSCAL_DIV, PWM_COUNT_US(1000, PWM_PSCAL_DIV), PWM_COUNT_US(500, PWM_PSCAL_DIV));

__STATIC_INLINE void pwm_enable (enum PWM_CH *pwmch*, uint32_t *able*)

Enable or disable pwm.

Parameters:

in	*pwmch*	PWM_CH0 or PWM_CH1
in	*able*	MASK_ENABLE or MASK_DISABLE

Description

This function is used to enable or disable the specified PWM channel

__STATIC_INLINE void pwm_clock_on (void)

Enable PWM module clock.

Description

This function is used to enable PWM module clock

__STATIC_INLINE void pwm_clock_off (void)

Disable PWM module clock.

Description

This function is used to disable PWM module clock

2.7 RTC Driver

Detailed Description

QN9020 Real Time Clock (RTC) module provides user the real time and calendar message. The RTC real time is based on external or internal low power 32 KHz clock, and its features are listed as follow:

15-bit counter to generate second with calibration function
Operate on external/internal 32 KHz clock
Provide 32-bit second counter
Programmable second and capture interrupts generation
Support input capture function with programmable noise canceller
The capture edge can be configured as positive or negative edge
Capture interrupt can be masked

Data Structure Documentation

struct rtc_time

Data Fields:

uint8_t	hour	Hour
uint8_t	minute	Minute
uint8_t	second	Second

struct rtc_date

Data Fields:

uint8_t	year	Year, start from 2000 year
uint8_t	month	Month
uint8_t	day	Day
uint8_t	week	[0-6] : 0:sunday 1:monday .....

struct rtc_env_tag

Data Fields:

struct rtc_time time
struct rtc_date date
void(* callback)(void)

Field Documentation

struct rtc_time rtc_env_tag::time

RTC time structure

struct rtc_date rtc_env_tag::date

RTC date structure

void(* rtc_env_tag::callback)(void)

The callback of RTC interrupt

Macro Definition Documentation

#define SECONDINDAY 86400

Total seconds in a day.

#define SECONDINHOUR 3600

Total seconds in an hour.

#define DAYINYEAR 365

Total days in a year.

#define DAYINLEAPYEAR 366

Total days in a leap year.

#define DAYINBIGMONTH 31

Total days in a big month.

#define DAYINLITTLEMONTH 30

Total days in a little month.

Enumeration Type Documentation

enum RTC_CAP_EDGE

RTC capture edge selection.

Enumerator:

RTC_CAP_EDGE_POS Rising edge is set as RTC capture edge

RTC_CAP_EDGE_NEG Falling edge is set as RTC capture edge

Function Documentation

uint8_t dec2bcd (uint8_t *decade*)

Decade convert to BCD

Parameters:

in	*decade*	0 ~ 99

Returns:

bcd BCD code

uint8_t bcd2dec (uint8_t *bcd*)

BCD convert to decade

Parameters:

in	*bcd*	BCD code

Returns:

decade 0 ~ 99

void RTC_IRQHandler (void)

Real time clock interrupt handler

void RTC_CAP_IRQHandler (void)

Real time clock capture interrupt handler

void rtc_init (void)

Real time clock initialization

Description

Initial RTC environment variable, it consists of clear callback function pointer.

void rtc_calibration (uint8_t *dir*, uint16_t ppm)

RTC calibration.

Parameters:

in	*dir*	direction of calibration
in	*ppm*	part per million

Description

This function is used to calibrate RTC, and should be called before setting time.

void rtc_correction (uint32_t *sleep_count*)

RTC correction.

Parameters:

in	*sleep_count*	add sleep count to RTC counter

Description

This function is used to correct RTC time after CPU wakeup

void rtc_capture_enable (enum RTC_CAP_EDGE edge, void(*)(void) callback)

Enable RTC capture

Parameters:

in	*edge*	RTC captrue edge selection: posedge or negedge
in	*callback*	callback function

Description

This function is used to initialize and enable RTC capture mode.

void rtc_capture_disable (void)

Disable RTC capture

Description

This function is used to disable RTC captrue function

void rtc_time_set (uint8_t year, uint8_t month, uint8_t day, uint8_t hour, uint8_t minute, uint8_t second, void(*)(void) callback)

Update RTC time.

Parameters:

in	*year*	base on 2000 eg. if the year is 2012, the param year = 12
in	*month*	1 ~ 12
in	*day*	1 ~ 31
in	*hour*	0 ~ 23
in	*minute*	0 ~ 59
in	*second*	0 ~ 59
in	*callback*	callback function

Description

The function is used to set RTC date, time and install callback function

static void rtc_time_parse (uint32_t time)[static]

Get current RTC time.

Parameters:

in	*time*	total seconds start form 1970.01.01, 00:00:00

Description

This function is used to parse RTC counter value to date and time

void rtc_time_get (void)

Get current RTC time.

Description

This function is used to get current RTC time.

__STATIC_INLINE void rtc_int_enable (void)

Enable RTC interrupt.

Description

This function is used to enable RTC interrupt

__STATIC_INLINE void rtc_int_disable (void)

Disable RTC interrupt.

Description

This function is used to disable RTC interrupt

__STATIC_INLINE void rtc_clock_on (void)

Enable RTC module clock.

Description

This function is used to enable RTC module clock

__STATIC_INLINE void rtc_clock_off (void)

Disable RTC module clock.

Description

This function is used to disable RTC module clock

__STATIC_INLINE void rtc_reset (void)

Reset RTC module.

Description

This function is used to reset RTC module

Variable Documentation

struct rtc_env_tag rtc_env = {0}

RTC environment variable.

struct rtc_capture_env_tag rtc_capture_env = {0}

RTC Capture environment variable.

struct rtc_env_tag rtc_env

RTC environment variable.

2.8 Serial Flash Driver

Detailed Description

QN9020 contains a Serial Flash Controller, which has mainly 2 functions: access external serial flash (erase/read/write) and boot from external serial flash (copy code from external serial flash to internal RAM and then to execute). The main features are listed as follow:

Access serial flash by SPI master port
Support SPI mode 0, up to 16MHz clock output
Flash command configurable
Two 8-bit shift registers for data transmit and receive
6 bit pre-scaler
Boot error detection
Support code encryption & decryption

Macro Definition Documentation

#define FLASH_CLK_DIV(x) (g_AhbClock/(2*(x)) - 1)

Set serial Flash clock divider.

#define RD_FLASH_ST_CMD (g_flash_cmd[RDSR_CMD] << 8)

Read serial flash status command.

#define FLASH_CMD_RDID 0x9F

RDID (Read Identification)

#define FLASH_CMD_RES 0xAB

RES (Read Electronic ID)

#define FLASH_CMD_REMS 0x90

REMS (Read Electronic & Device ID)

#define FLASH_CMD_WRSR 0x01

WRSR (Write Status Register)

#define FLASH_CMD_RDSR 0x05

RDSR (Read Status Register)

#define FLASH_CMD_READ 0x03

READ (1 x I/O)

#define FLASH_CMD_FASTREAD 0x0B

FAST READ (Fast read data)

#define FLASH_CMD_DREAD 0x3B

DREAD (1In/2 Out fast read)

#define FLASH_CMD_WREN 0x06

WREN (Write Enable)

#define FLASH_CMD_WRDI 0x04

WRDI (Write Disable)

#define FLASH_CMD_PP 0x02

PP (page program)

#define FLASH_CMD_SE 0x20

SE (Sector Erase)

#define FLASH_CMD_BE 0xD8

BE (Block Erase)

#define FLASH_CMD_CE 0x60

CE (Chip Erase) hex application: 60 or C7

#define FLASH_CMD_DP 0xB9

DP (Deep Power Down)

#define FLASH_CMD_RDP 0xAB

RDP (Release form Deep Power Down)

Enumeration Type Documentation

enum FLASH_CMD

Serial flash command index.

Enumerator:

RDSR_CMD read status register

WREN_CMD flash write enable

SE_CMD sector erase flash

BE_CMD block erase flash

CE_CMD erase whole flash

DPD_CMD deep power down

RDPD_CMD release deep power down

RESERVED RESERVED

MAX_FLASH_CMD_NUM it must 4 integer times

enum POWER_TYPE

Serial flash power type.

Enumerator:

FLASH_POWER_DOWN Set Serial flash power down

FLASH_POWER_ON Set Serial flash power on

Function Documentation

static bool is_flash_busy (void)[static]

Check whether flash is busy.

Returns:

flash write operateion status

Description

Check flash status register's WIP before program, erase or write status register

static void flash_write_enable (void)[static]

Enable Flash write operation.

void set_flash_clock (uint32_t *clock_div*)

Set Serial Flash controller clock.

Parameters:

in	*clock_div*	FLASH_CLK_DIV(clock), clock units is Hz

Description

This functin is used to set serial flash controller work clock.

uint32_t read_flash_id (void)

Read out flash ID.

Returns:

Flash ID

Description

This function is used to read serial flash ID, which consists of 3 or 4 bytes depending on difference vendor.

void sector_erase_flash (uint32_t *addr*, uint32_t *n*)

Erase n sectors of flash.

Parameters:

in	*addr*	A23-A0 specified a valid 24bit address of a sector
in	*n*	number of sector

Description

This function is used to erase serial flash sector.

void block_erase_flash (uint32_t *addr*, uint32_t *block_size*, uint32_t *n*)

Erase n blocks of flash.

Parameters:

in	*addr*	A23-A0 specified a valid 24bit address of a block.
in	*block_size*	flash a block content size
in	*n*	requirement erasing number blocks

Description

This function is used to erase serial flash block.

void read_flash (uint32_t *addr*, uint32_t * *pBuf*, uint32_t *nByte*)

Read data form flash.

Parameters:

in	*addr*	flash address(3 bytes)
in	*pBuf*	pointer to read data buffer address
in	*nByte*	read size, it must <= 256 and must be 4 integer times

Description

This function is used to read data from serial flash.

Note:

1. The parameter "addr" note:
   - When the address range is from 0x00 to 0x1000 (NVDS area), the address must be 4 integer times.
   - When the address range is greater than or equal to 0x1000 (Code area), the address must be 256 integer times. (Encryption request)
2. The parameter "nByte" note:
   - When the address range is from 0x00 to 0x1000 (NVDS area), the size must be 4 integer times and less than or equal to 256.

- When the address range is greater than or equal to 0x1000 (Code area), the size must be 256 bytes integer times. (Encryption request)

void write_flash (uint32_t *addr*, const uint32_t * *pBuf*, uint32_t *nByte*)

Write data to flash.

Parameters:

in	*addr*	flash address(3 bytes)
in	*pBuf*	pointer to write data address
in	*nByte*	write size, it must <= 256 and must be 4 integer times

Description

This function is used to write data to serial flash.

Note:

1. The parameter "addr" note:
   - When the address range is from 0x00 to 0x1000 (NVDS area), the address must be 4 integer times.
   - When the address range is greater than or equal to 0x1000 (Code area), the address must be 256 integer times. (Encryption request)
2. The parameter "nByte" note:
   - When the address range is from 0x00 to 0x1000 (NVDS area), the size must be 4 integer times and less than or equal to 256.
   - When the address range is greater than or equal to 0x1000 (Code area), the size must be 256 bytes. (Encryption request)

bool is_flash_present (void)

check whether flash is present

Returns:

ture or false

void power_on_flash (void)

Power on serial flash.

void power_off_flash (void)

Power off serial flash.

__STATIC_INLINE void flash_clock_on (void)

Enable Serial Flash module clock.

Description

This function is used to enable Serial Flash module clock

__STATIC_INLINE void flash_clock_off (void)

Disable Serial Flash module clock.

Description

This function is used to disable Serial Flash module clock

Variable Documentation

uint8_t g_flash_cmd[MAX_FLASH_CMD_NUM]

```
Initial value:=
{
    0x05,
    0x06,
```



```
    0x20,
    0x52,
    0x60,
    0xB9,
    0xAB,
    0x01,
}
```

Serial flash command list.

uint8_t g_flash_cmd[]

Serial flash command list.

2.9 SPI Driver

Detailed Description

The Serial Peripheral Interface (SPI) is a synchronous serial data communication protocol which operates in full duplex mode. Devices communicate in master/slave mode with 4-wire bi-direction interface. QN9020 contains 2 sets of SPI controller performing a serial-to-parallel conversion on data received from a peripheral device, and a parallel-to-serial conversion on data transmitted to a peripheral device. Each SPI set can drive up to 2 external peripherals. It also can be driven as the slave device when the slave mode is enabled.

Each controller can generate an individual interrupt signal when data transfer buffer is empty or receive buffer is full. The active level of device/slave select signal can be programmed to low active or high active, which depends on the connected peripheral. Writing a divisor into DIVIDER register can program the frequency of serial clock output when it is as the master.

The main features of SPI are listed as follow:

2 sets of SPI controller
Support master/slave mode operation
2 slave/device select lines in the master mode
Variable output serial clock frequency in master mode
SPI mode 0/1/2/3 configurable
8 bit/ 32 bit data width configurable
MSB or LSB first data transfer
4 bytes TX & RX synchronies FIFO
Both TX & RX DMA request

Data Structure Documentation

struct spi_txrxchannel

Data Fields

- int32_t size
- uint8_t * bufptr
- void(* callback)(void)

Field Documentation

int32_t spi_txrxchannel::size

Transmission size

uint8_t* spi_txrxchannel::bufptr

Data buffer of TX or RX

void(* spi_txrxchannel::callback)(void)

Callback at the end of transmission

struct spi_env_tag

Data Fields:

enum SPI_MODE	mode	SPI work mode
enum SPI_BUFFER_WIDTH	width	SPI transmission width
struct spi_txrxchannel	tx	Instance of TX
struct spi_txrxchannel	rx	Instance of RX

Macro Definition Documentation

#define SPI_SSx_CFG SPI_MASK_MSTR_SS0

SPI SS configure.

#define SPI_DUMMY_DATA (0xFFFFFFFF)

Dummy byte.

#define SPI_BITRATE(x) ((__USART_CLK/(2*(x)) - 1) << SPI_POS_CLK_DIV_MASTER)

SPI bit rate algorithm, x should less than (__USART_CLK/4)

#define SPI0_MOD_3WIRE_EN FALSE

SPI0 mode confiure: TRUE(3-wire: CS, CLK, DAT(I/O)), FALSE(4-wire: CS, CLK, DIN(I), DAT(O))

#define SPI1_MOD_3WIRE_EN FALSE

SPI1 mode confiure: TRUE(3-wire: CS, CLK, DAT(I/O)), FALSE(4-wire: CS, CLK, DIN(I), DAT(O))

Enumeration Type Documentation

enum SPI_POLARITY

SPI SCK polarity.

Enumerator:

SPI_CPOL_0 Set SPI SCK to low level at IDLE

SPI_CPOL_1 Set SPI SCK to high level at IDLE

enum SPI_PHASE

SPI SCK phase.

Enumerator:

SPI_CPHA_0 Set SCK phase, 1ST EDGE

SPI_CPHA_1 Set SCK phase, 2ND EDGE

enum SPI_MODE

SPI module mode: master or slave.

Enumerator:

SPI_MASTER_MOD Set SPI mode to master mode

SPI_SLAVE_MOD Set SPI mode to slave mode

enum SPI_BIT_ORDERING

SPI Bit ordering.

Enumerator:

SPI_LSB_FIRST Send LSB first

SPI_MSB_FIRST Send MSB first

enum SPI_BUFFER_WIDTH

SPI buffer width.

Enumerator:

SPI_8BIT Set SPI buffer width is 8 bits

SPI_32BIT Set SPI buffer width is 32 bits

enum SPI_BYTE_ENDIAN

SPI byte endian.

Enumerator:

SPI_BIG_ENDIAN Set SPI as big endian mode

SPI_LITTLE_ENDIAN Set SPI as little endian mode

enum SPI_TX_STATE

SPI TX status.

Enumerator:

SPI_TX_BUF_BUSY SPI TX busy

SPI_LAST_BYTE_ONGOING SPI last byte ongoing

SPI_TX_FREE SPI Tx free

enum SPI_INT_TYPE

SPI interrupt type.

Enumerator:

SPI_TX_INT SPI TX interrupt

SPI_RX_INT SPI RX interrupt

Function Documentation

static void spi_transmit_data (QN_SPI_TypeDef * *SPI*, struct spi_env_tag * *spi_env*)[static]

Transmit data to SPI TX FIFO.

Parameters:

in	*SPI*	QN_SPI0 or QN_SPI1
in	*spi_env*	Environment variable of specified SPI port

Description

Start to transmit date to specified SPI port until expected tramsmitting data size is reduced to zero.

static void spi_receive_data (QN_SPI_TypeDef * *SPI*, struct spi_env_tag * *spi_env*)[static]

Receives data from SPI RX FIFO.

Parameters:

in	*SPI*	QN_SPI0 or QN_SPI1
in	*spi_env*	Environment variable of specified SPI port

Description

Start to receive date from specified SPI port until expected receiving data size is reduced to zero.

void SPI0_IRQHandler (void)

SPI0 RX interrupt handler.

Description

If SPI0 RX FIFO is not empty, it then generates interrupt. In this handler, data is received from port SPI0 until expected receiving data size is reduced to zero. After last data received, the callback function is called.

void spi_init (QN_SPI_TypeDef * *SPI*, uint32_t *bitrate*, enum SPI_BUFFER_WIDTH *width*, enum SPI_MODE *mode*)

Initialize the SPI.

Parameters:

in	*SPI*	QN_SPI0 or QN_SPI1
in	*bitrate*	sck speed: SPI_BITRATE(1000) means 1Kbps
in	*width*	32bits or 8bits
in	*mode*	master or slave

Description

This function is used to initialize SPI. It consists of bit rate, transmit width, SPI mode, big/little endian, MSB/LSB first, master/salve. The function is also used to enable specified SPI interrupt, and enable NVIC SPI IRQ.

void spi_read (QN_SPI_TypeDef * *SPI*, uint8_t * *bufptr*, int32_t *size*, void(*)(void) *rx_callback*)

Start a data reception.

Parameters:

in	*SPI*	QN_SPI0 or QN_SPI1
in,out	*bufptr*	Pointer to the RX data buffer
in	*size*	Size of the expected reception, must be multiple of 4 at 32bit mode
in	*rx_callback*	Callback for end of reception

Description

This function is used to read Rx data from RX FIFO and the data will be stored in bufptr. As soon as the end of the data transfer or a buffer overflow is detected, the callback function is called.

void spi_write (QN_SPI_TypeDef * *SPI*, uint8_t * *bufptr*, int32_t *size*, void(*)(void) *tx_callback*)

Start a data transmission.

Parameters:

in	*SPI*	QN_SPI0 or QN_SPI1
in	*bufptr*	Pointer to the TX data buffer
in	*size*	Size of the transmission, must be multiple of 4 at 32bit mode
in	*tx_callback*	Callback for end of transmission

Description

This function is used to write data into TX buffer to transmit data by SPI. As soon as the end of the data transfer is detected, the callback function is called.

int spi_check_tx_free (QN_SPI_TypeDef * *SPI*)

Check if tx is ongoing.

Returns:

spi tx/rx status

Parameters:

in	*SPI*	QN_SPI0 or QN_SPI1

Description

This function is used to check SPI TX status

__STATIC_INLINE void spi_clock_on (QN_SPI_TypeDef * *SPI*)

Enable SPI module clock.

Parameters:

in	*SPI*	QN_SPI0 or QN_SPI1

Description

This function is used to enable SPI module clock

__STATIC_INLINE void spi_clock_off (QN_SPI_TypeDef * *SPI*)

Disable SPI module clock.

Parameters:

in	*SPI*	QN_SPI0 or QN_SPI1

Description

This function is used to disable SPI module clock

__STATIC_INLINE void spi_tx_data (QN_SPI_TypeDef * *SPI*, struct spi_env_tag * *spi_env*)

Send data to SPI.

Parameters:

in	*SPI*	QN_SPI0 or QN_SPI1
in	*spi_env*	Environment variable of specified SPI port

Description

Send a byte/word data to SPI

__STATIC_INLINE void spi_rx_data (QN_SPI_TypeDef * *SPI*, struct spi_env_tag * *spi_env*)

Get data form SPI.

Parameters:

in	*SPI*	QN_SPI0 or QN_SPI1
in	*spi_env*	Environment variable of specified SPI port

Description

Receive a byte/word data from SPI.

__STATIC_INLINE void spi_int_enable (QN_SPI_TypeDef * *SPI*, uint32_t *type*, uint32_t *able*)

Enable/Disable SPI interrupt.

Parameters:

in	*SPI*	QN_SPI0 or QN_SPI1
in	*type*	Interrupt type: refer to enum SPI_INT_TYPE
in	*able*	MASK_ENABLE or MASK_DISABLE

Description

Enable or disable specified SPI interrupt

Variable Documentation

struct spi_env_tag spi0_env

SPI0 environment variable.

struct spi_env_tag spi1_env

SPI1 environment variable.

struct spi_env_tag spi0_env

SPI0 environment variable.

struct spi_env_tag spi1_env

SPI1 environment variable.

2.10 Timer Driver

Detailed Description

QN9020 has two 32-bit timers Timer0/1, and two 16-bit timers Timer2/3. All the Timers support four operation modes, which allow user to easily implement a counting scheme. The Timers can perform functions like frequency measurement, event counting, interval measurement, clock generation, delay timing, and so on. The Timers also can generate an interrupt signal upon timeout, or provide the current value of count during operation, and support external count and capture functions.

The Features of Timer0/1/2/3 are listed as follow:

- 32/16-bit up counter timer with a 10-bit programmable prescaler
- Programmable clock sources, PCLK, 32KHz and external input
- Support four operation modes, which are free running mode, input capture timer mode, input capture event mode input capture counter mode
- Free running timer mode
  - Programmable interrupt period by setting the TOP register
  - Generate compare interrupt if the interrupt is enabled
  - Generate PWM waveform if PWM output enable (PWM_OE) bit is set
  - Programmable PWM period, duty, and PWM polarity
- Input capture timer mode
  - Pulse width, duty and period measurement
  - Capture on either positive or negative edge or both
  - Optional digital noise filtering on capture input
  - Programmable interrupt generation
- Input capture event mode
  - 16/8-bit event counter
- Input capture counter mode
  - 16/8-bit event number register shared with TOP register

Data Structure Documentation

struct timer_env_tag

Data Fields

uint32_t count
void(* callback)(void)

Field Documentation

uint32_t timer_env_tag::count

Timer counter value, different working modes have different values

void(* timer_env_tag::callback)(void)

The callback of timer interrupt

Macro Definition Documentation

#define TIMER_CLK(x) __TIMER_CLK

TIMER_CLK = AHB_CLK / (2*(TIMER_DIV + 1))

#define PSCL_DIV(n) ((n) + 1)

Timer prescaler divider algorithm

#define TIMER_DIV 0x1

Default timer divider value is 0x01

#define TIMER_PSCAL_DIV 0x3

Set prescaler divider value

#define TIMER_COUNT_S(s, pscl_div) ((s) * PSCL_CLK(TIMER_DIV, pscl_div))

Set timer counter top value (TOPR = delayInS * (PSCL_CLK))

#define TIMER_COUNT_MS(ms, pscl_div) ((ms) * (PSCL_CLK(TIMER_DIV, pscl_div) / 1000))

Set timer counter top value (TOPR = delayInMs * (PSCL_CLK / 1000))

#define TIMER_COUNT_US(us, pscl_div) ((us) * (PSCL_CLK(TIMER_DIV, pscl_div) / 1000000))

Set timer counter top value (TOPR = delayInUs * (PSCL_CLK / 1000000))

#define FREE_RUNNING_MOD (0 << TIMER_POS_OMS)

Timer mode: free running mode, using for normal timer function, PWM(compare counter)

#define INCAP_TIMER_MOD (1 << TIMER_POS_OMS)

Timer mode: input capture timer mode, using for measure period, pulse.

#define INCAP_EVENT_MOD (2 << TIMER_POS_OMS)

Timer mode: input capture event mode, using for measure event number within fix time.

#define INCAP_COUNTER_MOD (3 << TIMER_POS_OMS)

Timer mode: input capture counter mode, using for measure time within fix number event.

#define TIMER_PWM_POL_CFG 0

Timer PWM output polarity configuration

#define TIMER_INCAP_PIN_CFG INCAP_PIN0

Timer input capture pin configuration

Enumeration Type Documentation

enum TIMER_CSS

Timer clock source

Enumerator:

CLK_EXT Set timer clock source is external clock

CLK_ANCMP_OUT Set timer clock source is analog comparator output

CLK_PSCL Set timer clock source is timer prescaler

enum INCAP_PIN

Input capture PIN

Enumerator:

INCAP_PIN0 Set input captrue pin to PIN0

INCAP_PIN1 Set input captrue pin to PIN1

INCAP_PIN2 Set input captrue pin to PIN2

INCAP_PIN3 Set input captrue pin to PIN3

enum INCAP_EDGE

Input capture edge type

Enumerator:

INCAP_EDGE_POS Rising edge is set as input capture edge

INCAP_EDGE_NEG Falling edge is set as input capture edge

INCAP_EDGE_BOTH Both rising and falling edge are set as input capture edge

enum INCAP_SOURCE

Input capture source

Enumerator:

INCAP_SRC_PIN Set input capture source is GPIO pin

INCAP_SRC_ANCMP Set input capture source is analog comparator output

enum CMP_PWM_POL

PWM output default level

Enumerator:

CMP_PWM_POL_H Set timer pwm output default level is high level

CMP_PWM_POL_L Set timer pwm output default level is low level

Function Documentation

void timer_delay (QN_TIMER_TypeDef * *TIMER*, uint32_t *pscal*, uint32_t *count*)

Start the timer delay in micro seconds, until elapsed.

Parameters:

in	*TIMER*	QN_TIMER0,1,2,3
in	*pscal*	timer prescaler value
in	*count*	counter value

Description

This function is used to do precise time delay. HOW TO SET? e.g:
__AHB_CLK = 16000000 (16MHz), TIMER_DIV = 1, ==> TIMER_CLK = 4000000(4MHz)
PSCL_DIV = 3, ==> PSCL_CLK = 1000000Hz
delayInUs range: 1us - 4294967295 us (32bit)
delayInUs range: 1us - 65535 us (16bit)

timer_delay(QN_TIMER0, 3, TIMER_COUNT_US(100, 3)); // timer delay 100us

void TIMER0_IRQHandler (void)

Timer0 interrupt handler.

void TIMER1_IRQHandler (void)

Timer1 interrupt handler.

void TIMER2_IRQHandler (void)

Timer2 interrupt handler.

void TIMER3_IRQHandler (void)

Timer3 interrupt handler.

void timer_init (QN_TIMER_TypeDef * *TIMER*, void(*)(void) *callback*)

Initialize the timer.

Parameters:

in	*TIMER*	QN_TIMER0,1,2,3
in	*callback*	Call back function name for specified interrupt event

Description

Initialize the timer module.

void timer_config (QN_TIMER_TypeDef * *TIMER*, uint32_t *pscal*, uint32_t *count*)

Configure the timer.

Parameters:

in	*TIMER*	QN_TIMER0,1,2,3
in	*pscal*	timer prescaler value
in	*count*	counter value

Description

Configure the timer to work in timer mode, with this function users can easily set Timer pre-scaler, and count number.

void timer_pwm_config (QN_TIMER_TypeDef * *TIMER*, uint32_t *pscal*, uint32_t *periodcount*, uint32_t *pulsecount*)

Configure the timer pwm function.

Parameters:

in	*TIMER*	QN_TIMER0,1,2,3
in	*pscal*	timer prescaler value
in	*periodcount*	count value of period
in	*pulsecount*	count value of pulse

Description

Configure the timer to work in PWM mode, with this function users can easily set Timer pre-scaler, period, and pulse width.

void timer_capture_config (QN_TIMER_TypeDef * *TIMER*, uint32_t *cap_mode*, uint32_t *pscal*, uint32_t *count*, uint32_t *event_num*)

Configure timer capture function.

Parameters:

in	*TIMER*	QN_TIMER0, QN_TIMER1, QN_TIMER2, QN_TIMER3
in	*cap_mode*	INCAP_TIMER_MOD, INCAP_EVENT_MOD, INCAP_COUNTER_MOD
in	*pscal*	timer prescaler value
in	*count*	count value, active in INCAP_EVENT_MOD
in	*event_num*	active in INCAP_COUNTER_MOD

Description

Configure the timer to work in capture mode, with this function users can easily set input capture mode, Timer pre-scaler, and count/event number.

__STATIC_INLINE void timer_enable (QN_TIMER_TypeDef * *TIMER*, uint32_t *able*)

Enable or disable timer.

Parameters:

in	*TIMER*	QN_TIMER0,1,2,3
in	*able*	MASK_ENABLE or MASK_DISABLE

Description

This function is used to enable or disable the specified Timer.

__STATIC_INLINE void timer_clock_on (QN_TIMER_TypeDef * *TIMER*)

Enable TIMER module clock.

Parameters:

in	*TIMER*	QN_TIMER0,1,2,3

Description

This function is used to enable TIMER module clock

__STATIC_INLINE void timer_clock_off (QN_TIMER_TypeDef * *TIMER*)

Disable TIMER module clock.

Parameters:

in	*TIMER*	QN_TIMER0,1,2,3

Description

This function is used to disable TIMER module clock

__STATIC_INLINE void timer_reset (QN_TIMER_TypeDef * *TIMER*)

Reset TIMER module.

Parameters:

in	*TIMER*	QN_TIMER0,1,2,3

Description

This function is used to reset TIMER module

Variable Documentation

struct timer_env_tag timer0_env

TIMER0 environment variable.

struct timer_env_tag timer1_env

TIMER1 environment variable.

struct timer_env_tag timer2_env

TIMER2 environment variable.

struct timer_env_tag timer3_env

TIMER3 environment variable.

2.11 UART Driver

Detailed Description

The Universal Asynchronous Receiver/Transmitter (UART) performs a serial-to-parallel conversion on data characters received from the peripheral, and a parallel-to-serial conversion on data characters received from the CPU. QN9020 UART is an AMBA slave module that connects to the Advanced Peripheral Bus (APB), and the features are listed as fellow:

- Compliance to the AMBA specification (Rev 2.0, APB4)
- Configurable full-duplex or half-duplex data transmission
- Configurable hardware flow control with nRTS and nCTS option
- Receive and transmit data buffer are supported (only one depth)
- Configurable over-sampling rate (8 or 16)
- Programmable baud rate generator, baud rates up to 2MHz if 16MHz UART clock is adopted
- Full programmable serial interface characteristics:
  - Data width support 8bit
  - Odd, even or no-parity bit generation and detection
  - 1 or 2 stop bit generation
  - Configurable LSB- or MSB-first transfer
- Parity¡¢overrun and framing error detection
- Transmit and receive interrupts
- Support for Direct Memory Access(DMA)
- Line-break generation and detection

Macro Definition Documentation

#define USARTx_CLK(div) __USART_CLK

USARTx_CLK = AHB_CLK / (2*(USARTx_DIVIDER + 1))

Enumeration Type Documentation

enum UART_OVERSAMPLE_TYPE

UART oversample type.

Enumerator:

UART_OVS8 Set oversampling is 8

UART_OVS16 Set oversampling is 16

enum UART_BAUDRATE

UART buadrate.

Enumerator:

UART_1200 Set baud rate to 1200 when UART clock is 8MHz

UART_2400 Set baud rate to 2400 when UART clock is 8MHz

UART_4800 Set baud rate to 4800 when UART clock is 8MHz

UART_9600 Set baud rate to 9600 when UART clock is 8MHz

UART_14400 Set baud rate to 14400 when UART clock is 8MHz

UART_19200 Set baud rate to 19200 when UART clock is 8MHz

UART_28800 Set baud rate to 28800 when UART clock is 8MHz

UART_38400 Set baud rate to 38400 when UART clock is 8MHz

UART_57600 Set baud rate to 57600 when UART clock is 8MHz

UART_64000 Set baud rate to 64000 when UART clock is 8MHz
UART_76800 Set baud rate to 76800 when UART clock is 8MHz
UART_115200 Set baud rate to 115200 when UART clock is 8MHz
UART_128000 Set baud rate to 128000 when UART clock is 8MHz
UART_230400 Set baud rate to 230400 when UART clock is 8MHz
UART_345600 Set baud rate to 345600 when UART clock is 8MHz
UART_460800 Set baud rate to 460800 when UART clock is 8MHz
UART_500000 Set baud rate to 500000 when UART clock is 8MHz

enum UART_TX_STATE

UART TX status.

Enumerator:

UART_TX_BUF_BUSY Uart TX busy
UART_LAST_BYTE_ONGOING Uart last byte ongoing
UART_TX_FREE Uart Tx free

Function Documentation

static void uart_transmit_data (QN_UART_TypeDef * *UART*, struct uart_env_tag * *uart_env*)[static]

Transmit data to UART TX FIFO.

Parameters:

in	*UART*	QN_UART0 or QN_UART1
in	*uart_env*	Environment Variable of specified UART port

Description

Start to transmit data to specified UART port until expected tramsmitting data size is decreased to zero.

static void uart_receive_data (QN_UART_TypeDef * *UART*, struct uart_env_tag * *uart_env*)[static]

Receive data from UART RX FIFO.

Parameters:

in	*UART*	QN_UART0 or QN_UART1
in	*uart_env*	Environment Variable of specified UART port

Description

Start to receive data from specified UART port until expected receiving data size is decreased to zero.

void UART0_TX_IRQHandler (void)

UART0 TX interrupt handler.

Description

Transmit data to port UART0 until expected tramsmitting data size is decreased to zero.

void UART0_RX_IRQHandler (void)

UART0 RX interrupt handler.

Description

Receive data from port UART0 until expected receiving data size is decreased to zero.

void uart_init (QN_UART_TypeDef * *UART*, uint32_t *uartclk*, enum UART_BAUDRATE *baudrate*)

Initialize the UART to default values.

Parameters:

in	*UART*	QN_UART0 or QN_UART1
in	*uartclk*	USARTx_CLK(div)
in	*baudrate*	baud rate

Description

This function is used to initialize UART, it consists of baud-rate, parity, data-bits, stop-bits, over sample rate and bit order. The function is also used to enable specified UART interrupt, and enable NVIC UART IRQ.

void uart_read (QN_UART_TypeDef * *UART*, uint8_t * *bufptr*, uint32_t *size*, void(*)(void) *rx_callback*)

Start a data reception.

Parameters:

in	*UART*	QN_UART0 or QN_UART1
in,out	*bufptr*	Pointer to the RX buffer
in	*size*	Size of the expected reception
in	*rx_callback*	Callback for end of reception

Description

This function is used to read Rx data from RX FIFO and the data will be stored in bufptr. As soon as the end of the data transfer is detected, the callback function is executed.

void uart_write (QN_UART_TypeDef * *UART*, uint8_t * *bufptr*, uint32_t *size*, void(*)(void) *tx_callback*)

Start a data transmission.

Parameters:

in	*UART*	QN_UART0 or QN_UART1
in	*bufptr*	Pointer to the TX data buffer
in	*size*	Size of the transmission
in	*tx_callback*	Callback for end of transmission

Description

This function is used to write data into TX buffer to transmit data by UART. As soon as the end of the data transfer is detected, the callback function is executed.

void uart_printf (QN_UART_TypeDef * *UART*, uint8_t * *bufptr*)

Send a string to UART.

Parameters:

in	*UART*	QN_UART0 or QN_UART1
in	*bufptr*	buffer pointer of tx data

Description

Print a string to specified UART port

void uart_finish_transfers (QN_UART_TypeDef * *UART*)

Wait until transfer finish.

Parameters:

in	*UART*	QN_UART0 or QN_UART1

Description

Waiting for specified UART port transfer finished

int uart_check_tx_free (QN_UART_TypeDef * *UART*)

Check if tx is ongoing.

Returns:

uart tx/rx status

Parameters:

in	*UART*	QN_UART0 or QN_UART1

Description

This function is used to check UART TX status

void uart_flow_on (QN_UART_TypeDef * *UART*)

Enable hardware flow control.

Parameters:

in	*UART*	QN_UART0 or QN_UART1

Description

Enable specified UART port hardware flow control

bool uart_flow_off (QN_UART_TypeDef * *UART*)

Disable hardware flow control.

Parameters:

in	*UART*	QN_UART0 or QN_UART1

Returns:

TRUE

Description

Disable specified UART port hardware flow control

__STATIC_INLINE uint8_t uart_read_one_byte (QN_UART_TypeDef * *UART*)

Get one byte form UART.

Parameters:

in	*UART*	QN_UART0 or QN_UART1

Returns:

uint8_t One byte data

Description

Receive 1 byte data from specified UART FIFO.

__STATIC_INLINE void uart_write_one_byte (QN_UART_TypeDef * *UART*, uint8_t *data*)

Send one byte to UART.

Parameters:

in	*UART*	QN_UART0 or QN_UART1
in	*data*	data which want to send

Description

Send 1 byte data from UART

__STATIC_INLINE void uart_rx_enable (QN_UART_TypeDef * *UART*, uint32_t *able*)

Enable/Disable UART RX.

Parameters:

in	*UART*	QN_UART0 or QN_UART1
in	*able*	MASK_ENABLE or MASK_DISABLE

Description

Enable or disable specified UART RX port

__STATIC_INLINE void uart_tx_enable (QN_UART_TypeDef * *UART*, uint32_t *able*)

Enable/Disable UART TX.

Parameters:

in	*UART*	QN_UART0 or QN_UART1
in	*able*	MASK_ENABLE or MASK_DISABLE

Description

Enable or disable specified UART TX port

__STATIC_INLINE void uart_rx_int_enable (QN_UART_TypeDef * *UART*, uint32_t *able*)

Enable/Disable all UART RX interrupt.

Parameters:

in	*UART*	QN_UART0 or QN_UART1
in	*able*	MASK_ENABLE or MASK_DISABLE

Description

Enable or disable specified UART RX interrupt

__STATIC_INLINE void uart_tx_int_enable (QN_UART_TypeDef * *UART*, uint32_t *able*)

Enable/Disable UART TX interrupt.

Parameters:

in	*UART*	QN_UART0 or QN_UART1
in	*able*	MASK_ENABLE or MASK_DISABLE

Description

Enable or disable specified UART TX interrupt

__STATIC_INLINE void uart_clock_on (QN_UART_TypeDef * *UART*)

Enable UART module clock.

Parameters:

in	*UART*	QN_UART0 or QN_UART1

Description

This function is used to enable UART module clock

__STATIC_INLINE void uart_clock_off (QN_UART_TypeDef * *UART*)

Disable UART module clock.

Parameters:

in	*UART*	QN_UART0 or QN_UART1

Description

This function is used to disable UART module clock

__STATIC_INLINE void usart_reset (uint32_t *usart*)

Reset USART module (UART&SPI)

Parameters:

in	*usart*	QN_UART0 / QN_UART1 / QN_SPI0 / QN_SPI1

Description

This function is used to reset USART module (include UART and SPI)

Variable Documentation

struct uart_env_tag uart0_env[static]

UART0 environment variable.

struct uart_env_tag uart1_env[static]

UART1 environment variable.

struct uart_divider_cfg uart_divider[UART_BAUD_MAX]

```
Initial value:=
{
    {0x01, 0xA0, 0x2B, },
    {0x00, 0xD0, 0x15, },
    {0x00, 0x68, 0x0B, },
    {0x00, 0x34, 0x05, },
    {0x00, 0x22, 0x2E, },
    {0x00, 0x1A, 0x03, },
    {0x00, 0x11, 0x17, },
    {0x00, 0x0D, 0x01, },
    {0x00, 0x08, 0x2C, },
    {0x00, 0x07, 0x34, },
    {0x00, 0x06, 0x21, },
    {0x00, 0x04, 0x16, },
    {0x00, 0x03, 0x3A, },
    {0x00, 0x02, 0x0B, },
    {0x00, 0x01, 0x1D, },
    {0x00, 0x01, 0x05, },
    {0x00, 0x01, 0x00, },
}
```

Description

HOW TO CONFIGURATE BAUD RATE?

If oversample is 16, the required baud rate is 230400 and UARTCLK = 8MHz, then: Baud Rate Divisor = (8*1000000)/(16*230400) = 2.170, This means BRDI = 2 and BRDF = 0.170, Therefore, fractional part, m = integer((0.170*64)+0.5) = 11.

If the required baud rate is 921600 and UARTCLK = 16MHz then: Baud Rate Divisor = (16*1000000)/(16*921600) = 1.085, This means BRDI = 1 and BRDF = 0.085, Therefore, fractional part, m = integer((0.085*64)+0.5) = 5.

2.12 WDT Driver

Detailed Description

The purpose of Watchdog Timer (WDT) is to perform a system reset after the software running into a problem. This prevents system from hanging for an infinite period of time. The main features of QN9020 WDT are listed as follow:

32-bit down counter with a programmable timeout interval
32KHz clock(WDOGCLK=PCLK, WDOGCLKEN=32K)
Interrupt output generation on timeout
Reset signal generation on timeout if the interrupt from the previous timeout remains unserviced by software
Lock register to protect registers from being altered by runaway software

Enumeration Type Documentation

enum WDT_MODE

Watchdog timer work mode

Enumerator:

WDT_NO_ACTION_MOD Set watchdog timer work at no action mode

WDT_INT_MOD Set watchdog timer work at interrupt mode

WDT_RESET_MOD Set watchdog timer work at reset mode

Function Documentation

void wdt_irq_clear (void)

Clear watchdog timer interrupt request.

void WDT_IRQHandler (void)

Watchdog timer interrupt handler.

void wdt_init (unsigned int *cycle*, enum WDT_MODE *mode*)

Watchdog timer initialization.

Parameters:

in	*cycle*	time-out interval
in	*mode*	wrok mode: WDT_NO_ACTION_MOD/WDT_INT_MOD/WDT_RESET_MOD

Description

This function is used to set WDT work mode and WDT time-out interval.

void wdt_set (unsigned int *cycle*)

Update watchdog timer counter.

Parameters:

in	*cycle*	time-out interval

Description

This function is used to set WDT time-out interval.

__STATIC_INLINE void wdt_clock_on (void)

Enable WDT module clock.

Description

This function is used to enable WDT module clock

__STATIC_INLINE void wdt_clock_off (void)

Disable WDT module clock.

Description

This function is used to disable WDT module clock

__STATIC_INLINE void wdt_reset (void)

Reset WDT module.

Description

This function is used to reset WDT module

__STATIC_INLINE void wdt_unlock (void)

Unlock watchdog timer access.

__STATIC_INLINE void wdt_lock (void)

Lock watchdog timer access.

Variable Documentation

volatile int reset_test = 0

Set to 1 during watchdog timer reset test so that WDT_IRQHandler will not clear the watchdog

2.13 Sleep Driver

Detailed Description

In QN9020, three sleep modes are defined according to cortex-M0 low power modes.

CPU clock gating mode: Cortex-M0 can be clock gated, NVIC remains sensitive to interrupts, all NVIC interrupt sources can wake up Cortex-M0.

CPU deep clock gating mode: Cortex-M0 and NVIC can be clock gated, WIC remains sensitive to selected interrupts, all WIC interrupt sources can wake up Cortex-M0, Cortex-M0 can be put into state retention.

CPU sleep mode: Power down Cotex-M0 processor, all clocks can be powered down, 32Khz clock is an option(if using sleep timer wakeup), WIC signals wake-up to PMU, all WIC interrupt sources can wake up Cortex-M0, Cortex-M0 can be put into state retention.

Data Structure Documentation

struct sleep_env_tag

Data Fields:

uint8_t	sleep_allow	
uint32_t	dev_active_bf	
bool	deep_sleep	
int	retention_modules	
int	wakeup_by_sleeptimer	

Macro Definition Documentation

#define PM_MASK_ADC_ACTIVE_BIT (0x00000001)

Device active bit field.

#define WAKEUP_BY_ALL_IRQ_SOURCE

```
Value:( WAKEUP_BY_GPIO          \
                              | WAKEUP_BY_ACMP0  \
                              | WAKEUP_BY_ACMP1  \
```



```
                                | WAKEUP_BY_BLE          \
                                | WAKEUP_BY_RTC_CAP      \
                                | WAKEUP_BY_RTC         \
                                | WAKEUP_BY_ADC         \
                                | WAKEUP_BY_DMA         \
                                | WAKEUP_BY_UART0_TX     \
                                | WAKEUP_BY_UART0_RX     \
                                | WAKEUP_BY_SPI0_TX      \
                                | WAKEUP_BY_SPI0_RX      \
                                | WAKEUP_BY_UART1_TX     \
                                | WAKEUP_BY_UART1_RX     \
                                | WAKEUP_BY_SPI1_TX      \
                                | WAKEUP_BY_SPI1_RX      \
                                | WAKEUP_BY_I2C         \
                                | WAKEUP_BY_TIMER0       \
                                | WAKEUP_BY_TIMER1       \
                                | WAKEUP_BY_TIMER2       \
                                | WAKEUP_BY_TIMER3       \
                                | WAKEUP_BY_WDT         \
                                | WAKEUP_BY_PWM0         \
                                | WAKEUP_BY_PWM1         \
                                | WAKEUP_BY_TUNER_SETTING)
```

Wakeup by all of the system interrupt source.

Typedef Documentation

typedef void(* p_rwble_prevent_sleep_set)(uint16_t prv_slp_bit)

OSC interrupt handler, BLE wakeup source.

Enumeration Type Documentation

enum POWER_MODE

power mode

Enumerator:

PM_ACTIVE CO_PD_DISALLOWED, disallow cpu clock off & cpu power down

PM_IDLE CPU_CLK_OFF_ALLOW

PM_SLEEP CPU_POWER_DOWN_ALLOW

PM_DEEP_SLEEP CPU_DEEP_SLEEP_ALLOW

enum SLEEP_MODE

QN9020 sleep mode.

Enumerator:

SLEEP_CPU_CLK_OFF Disable CPU clock

SLEEP_NORMAL Sleep

SLEEP_DEEP Deep Sleep

enum WAKEUP_SOURCE

QN9020 wakeup source.

Enumerator:

WAKEUP_BY_GPIO Wakeup by GPIO

WAKEUP_BY_ACMP0 Wakeup by ACMP0

WAKEUP_BY_ACMP1 Wakeup by ACMP1

WAKEUP_BY_BLE Wakeup by BLE
WAKEUP_BY_RTC_CAP Wakeup by RTC_CAP
WAKEUP_BY_OSC_EN Wakeup by OSC_EN
WAKEUP_BY_RTC Wakeup by RTC
WAKEUP_BY_ADC Wakeup by ADC
WAKEUP_BY_DMA Wakeup by DMA
WAKEUP_BY_UART0_TX Wakeup by UART0_TX
WAKEUP_BY_UART0_RX Wakeup by UART0_RX
WAKEUP_BY_SPI0_TX Wakeup by SPI0_TX
WAKEUP_BY_SPI0_RX Wakeup by SPI0_RX
WAKEUP_BY_UART1_TX Wakeup by UART1_TX
WAKEUP_BY_UART1_RX Wakeup by UART1_RX
WAKEUP_BY_SPI1_TX Wakeup by SPI1_TX
WAKEUP_BY_SPI1_RX Wakeup by SPI1_RX
WAKEUP_BY_I2C Wakeup by I2C
WAKEUP_BY_TIMER0 Wakeup by TIMER0
WAKEUP_BY_TIMER1 Wakeup by TIMER1
WAKEUP_BY_TIMER2 Wakeup by TIMER2
WAKEUP_BY_TIMER3 Wakeup by TIMER3
WAKEUP_BY_WDT Wakeup by WDT
WAKEUP_BY_PWM0 Wakeup by PWM0
WAKEUP_BY_PWM1 Wakeup by PWM1
WAKEUP_BY_CALIB Wakeup by CALIB
WAKEUP_BY_TUNER_RX Wakeup by TUNER_RX
WAKEUP_BY_TUNER_TX Wakeup by TUNER_TX
WAKEUP_BY_TUNER_SETTING Wakeup by TUNER_SETTING

Function Documentation

int usr_sleep (void)

Check application whether to enter sleep mode.

Returns:

sleep allowed status

void sleep_init (void)

Init sleep power down modules.

Description

This function is used to init MCU sleep mode.

void enter_sleep (enum SLEEP_MODE *mode*, uint32_t *iconfig*, void(*)(void) *callback*)

Enable sleep mode.

Parameters:

in	*mode*	sleep mode
in	*iconfig*	wakeup interrupt config

in	*callback*	callback after wakeup

Description

This function is used to set MCU into sleep mode, before enter sleep, wakeup source should be set.

void wakeup_by_gpio (enum gpio_pin *pin*, enum gpio_wakeup_type *type*)

Set GPIO wakeup.

Parameters:

in	*pin*	wakeup pin: P0 and P1
in	*type*	Wakeup type: high, low, change

Description

This function is used to set MCU wakeup by gpio pin.

void wakeup_by_analog_comparator (enum ACMP_CH *acmpch*, void(*)(void) *callback*)

Set analog comparator wakeup.

Parameters:

in	*acmpch*	enum ACMP_CH
in	*callback*	Callback function pointer, which is called in IRQHandler.

Description

This function is used to set MCU wakeup by analog comparator.

void wakeup_by_sleep_timer (int *clk_src*)

Set sleep timer wakeup.

Parameters:

in	*clk_src*	32KHz clock source

Description

This function is used to set MCU wakeup by sleep timer.

void sleep_cb (void)

Sleep wakeup callback function.

Description

This function will be called before clock switching to XTAL in sleep mode.

void enter_low_power_mode (uint32_t *en*)

Enter low power mode.

Parameters:

in	*en*	enabled peripheral at low power mode

Description

This function is used to set MCU entering into low power mode.

void restore_from_low_power_mode (void(*)(void) *callback*)

Restore from low power mode.

Parameters:

in	*callback*	callback before XTAL clock ready

Description

This function is used to set MCU restoring from low power mode, switch system clock to XTAL.

__STATIC_INLINE void exit_low_power_mode (void)

Exit low power mode.

Description

This function is used to set MCU exiting from low power mode, switch system clock to internal 20MHz.

__STATIC_INLINE void sleep_set_pm (uint8_t *pm*)

Set user program's power mode.

Parameters:

in	*pm*	active/clock off/sleep/deep sleep PM_ACTIVE PM_IDLE, PM_SLEEP, PM_DEEP_SLEEP

__STATIC_INLINE uint32_t sleep_get_pm (void)

Get user program's power mode.

Returns:

sleep allowed status

__STATIC_INLINE void dev_prevent_sleep (uint32_t *dev_bf*)

Device prevent sleep.

Parameters:

in	*dev_bf*	bit field of active device

__STATIC_INLINE void dev_allow_sleep (uint32_t *dev_bf*)

User device allow sleep.

Parameters:

in	*dev_bf*	bit field of active device

__STATIC_INLINE uint32_t dev_get_bf (void)

Get device bit field.

Returns:

device actived bits

2.14 System Controller Driver

Detailed Description

QN9020 System Controller mainly contains Reset Management Unit (RMU), Clock Management Unit (CMU) and Power Management Unit (PMU). The following functions are included in these units:

System registers management and module functional reset
Clock generator
System clock and peripherals clock
Low Power mode
PIN MUX

Macro Definition Documentation

#define AHB_CLK_DIV(n) (g_SystemClock/(2*n) - 1)

AHB_CLK = SYS_CLK/(2*(AHB_DIVIDER+1)), n is AHB_CLK;.

#define APB_CLK_DIV(n) (g_AhbClock/(2*n) - 1)

APB_CLK = AHB_CLK/(2*(APB_DIVIDER+1)), n is APB_CLK;.

#define TIMER_CLK_DIV(n) (g_AhbClock/(2*n) - 1)

TIMER_CLK = AHB_CLK/(2*(TIMER_DIVIDER+1)), n is TIMER_CLK;.

#define USARTx_CLK_DIV(n) (g_AhbClock/(2*n) - 1)

USARTx_CLK = AHB_CLK/(2*(USARTx_DIVIDER+1)), n is USARTx_CLK;.

#define BLE_CLK_DIV(n) (g_AhbClock/(2*n) - 1)

BLE_CLK = AHB_CLK/(2*(BLE_DIVIDER+1)), n is BLE_CLK;

Enumeration Type Documentation

enum CLK_MUX

Clock mux.

Enumerator:

CLK_XTAL External High frequency 16MHz or 32MHz

CLK_INT_20M 20MHz internal high frequency

CLK_INT_32M 32MHz PLL output

CLK_LOW_32K 32KHz low speed clock

enum RESET_CAUSE

Reset cause.

Enumerator:

NONE_RST Not reset or reset clear

POWER_ON_RST Power-on Reset (POR)

BROWN_OUT_RST Brown-out Detection (BOD)

EXT_PIN_RST RESET pin reset

WDT_RST Watchdog reset

LOCK_UP_RST ARM M0 Lockup signal output

REBOOT_RST Software triggered reset for system reboot

CPU_SYS_RST ARM M0 system reset requirement output

CPU_SOFT_RST CPU Software reset

enum CLK_TYPE

Clock type.

Enumerator:

XTAL_16M External XTAL frequency 16MHz

XTAL_32M External XTAL frequency 32MHz

PLL_32M Internal PLL 32MHz

INT_20M Internal 20MHz

RCO_32K 32KHz clock from RCO32

XTAL_32K 32KHz clock from XTAL32

enum MEM_BLOCK

Memory block.

Enumerator:

MEM_BLOCK0 Memory Block1: 0K ~ 8K

MEM_BLOCK1 Memory Block1: 8K ~ 16K

MEM_BLOCK2 Memory Block1: 16K ~ 24K

MEM_BLOCK3 Memory Block1: 24K ~ 32K

MEM_BLOCK4 Memory Block1: 32K ~ 40K

MEM_BLOCK5 Memory Block1: 40K ~ 48K

MEM_BLOCK6 Memory Block1: 48K ~ 56K

MEM_BLOCK7 Memory Block1: 56K ~ 64K

MEM_ALL Memory Block1: 56K ~ 64K

enum XTAL_CLK_SRC

XTAL clock source.

Enumerator:

CRYSTAL Use crystal oscillator between XTAL1/XTAL2

DIGIT_CLOCK Digital clock injection to XTAL1

SINGLE_SINE Single-end sine wave injection to XTAL1

DIFF_SINE Differential sine wave injection to XTAL1/XTAL2

Function Documentation

void syscon_set_sysclk_src (enum CLK_MUX *clk_src*, int *flag*)

set system clock source

Parameters:

in	*clk_src*	System clock source
in	*flag*	Indicating XTAL is 16MHz or 32MHz, or 32KHz is form XTAL32 or RCO32

Returns:

Description

This function is used to set system clock source.

void syscon_set_ahb_clk (int *clk*)

Set AHB clock.

Parameters:

in	*clk*	AHB clock frequency

Returns:

Description

This function is used to set AHB clock.

void syscon_get_ahb_clk (void)

Get AHB clock.

Description

This function is used to get AHB clock.

void syscon_set_apb_clk (int *clk*)

Set APB clock.

Parameters:

in	*clk*	APB clock frequency

Returns:

Description

This function is used to set APB clock.

void syscon_get_apb_clk (void)

Get APB clock.

Description

This function is used to get APB clock.

void syscon_set_timer_clk (int *clk*)

Set TIMER clock.

Parameters:

in	*clk*	TIMER clock frequency

Returns:

Description

This function is used to set TIMER clock.

void syscon_set_usart_clk (uint32_t *usart*, int *clk*)

Set USART clock.

Parameters:

in	*usart*	QN_UART0 / QN_UART1 / QN_SPI0 / QN_SPI1
in	*clk*	USART clock frequency

Returns:

Description

This function is used to set USART clock.

void syscon_set_ble_clk (int *clk*)

Set BLE clock.

Parameters:

in	*clk*	BLE clock frequency: only support 8M, 16M

Returns:

Description

This function is used to set BLE clock.

enum RESET_CAUSE syscon_get_reset_cause (void)

Get reset source.

Returns:

enum RESET_CAUSE

Description

This function is used to get system reset cause.

void syscon_enable_transceiver (uint32_t *able*)

Enable or disable transceiver.

Parameters:

in	*able*	MASK_ENABLE or MASK_DISABLE

Returns:

Description

This function is used to enable or disable transceiver, contains BLE clock setting and REF PLL power setting.

__STATIC_INLINE void clk32k_enable (int *flag*)

Enable 32K clock.

Parameters:

in	*flag*	XTAL_32K or RCO_32K

Returns:

Description

This function is used to enable 32K clock

__STATIC_INLINE void memory_power_off (int *memblk*, int *able*)

Power off memory enable or disable.

Parameters:

in	*memblk*	MEM_BLOCK1 ~ MEM_BLOCK7
in	*able*	MASK_ENABLE or MASK_DISABLE

Returns:

Description

This function is used to control memory power

__STATIC_INLINE void clk32k_power_off (int *flag*, int *able*)

Power off 32K clock enable or disable.

Parameters:

in	*flag*	XTAL_32K or RCO_32K
in	*able*	MASK_ENABLE or MASK_DISABLE

Returns:

Description

This function is used to control 32K clock power

__STATIC_INLINE void syscon_set_xtal_src (int *flag*, int *src*)

set XTAL clock source

Parameters:

in	*flag*	XTAL_32M or XTAL_32K
in	*src*	XTAL clock source

Returns:

Description

This function is used to set XTAL clock source.

2.15 Driver Configurations

Detailed Description

Driver Configurations define driver status (enable or disable), realization method (interrupt or polling), , which driver to use (dirver code or driver in ROM), driver callback status (enable or disable), and driver work mode (for example, I2C module work at MASTER or SLAVE mode). Users can modify these configurations.

The following is an example of how to configure UART driver:

CONFIG_ENABLE_DRIVER_UART: This macro can be set to TRUE or FALSE, means to enable or disable UART driver. Only if this macro value is TRUE, the other macros related to UART have meanings.

CONFIG_UART0_TX_DEFAULT_IRQHANDLER: This macro is used to enable or disable UART0 TX default interrupt request handler. It can be set to TURE or FALSE. If the macro is defined to FALSE, users can rewrite a new handler to replace the default handler. This macro will be effective under the condition of UART driver is enabled and UART0 TX interrupt is enabled.

CONFIG_UART0_TX_ENABLE_INTERRUPT: Define this macro to TRUE to enable UART0 TX interruption. Otherwise, UART0 data will be transmitted via polling.

CONFIG_ENABLE_ROM_DRIVER_UART: This macro set to TRUE means to use driver burned in ROM. All the UART APIs become to function pointer which point to ROM address and driver configurations are fixed. Otherwise, the UART source code will be used, and user can modify them.

UART_CALLBACK_EN: This macro means to enable or disable UART callback.

UART_BAUDRATE_TABLE_EN: This macro means to enable or disable UART baud rate parameters table, If the macro is defined to FALSE, baud rate will be set by formula calculation.

Macro Definition Documentation

#define __XTAL XTAL_16MHz

driver configuration

Extrenal frequency

#define __SYSTEM_CLOCK SYS_EXT_XTAL

System clock frequency

#define __AHB_CLK CLK_8M

AHB clock frequency

#define __APB_CLK CLK_8M

APB clock frequency

#define __BLE_CLK CLK_8M

BLE clock frequency

#define __TIMER_CLK CLK_8M

TIMER clock frequency

#define __USART_CLK CLK_8M

UART and SPI clock frequency

#define __32K_TYPE XTAL_32K

32K clock type: XTAL_32K, RCO_32K

#define CONFIG_ENABLE_DRIVER_GPIO TRUE

Enable/Disable GPIO Driver

#define CONFIG_GPIO_DEFAULT_IRQHANDLER TRUE

Enable/Disable GPIO Default IRQ Handler

#define CONFIG_GPIO_ENABLE_INTERRUPT TRUE

Enable/Disable GPIO Interrupt

#define CONFIG_ENABLE_DRIVER_SPI0 TRUE

Enable/Disable SPI Driver

#define CONFIG_SPI0_DEFAULT_IRQHANDLER TRUE

Enable/Disable SPI0 Default IRQ Handler

#define CONFIG_SPI0_TX_ENABLE_INTERRUPT TRUE

Enable/Disable(Polling) SPI0 TX Interrupt

#define CONFIG_SPI0_RX_ENABLE_INTERRUPT TRUE

Enable/Disable(Polling) SPI0 RX Interrupt

#define CONFIG_ENABLE_DRIVER_SPI1 TRUE

Enable/Disable SPI Driver

#define CONFIG_SPI1_DEFAULT_IRQHANDLER FALSE

Enable/Disable SPI1 Default IRQ Handler

#define CONFIG_SPI1_TX_ENABLE_INTERRUPT FALSE

Enable/Disable(Polling) SPI1 TX Interrupt

#define CONFIG_SPI1_RX_ENABLE_INTERRUPT FALSE

Enable/Disable(Polling) SPI1 RX Interrupt

#define CONFIG_ENABLE_DRIVER_UART0 TRUE

Enable/Disable UART Driver

#define CONFIG_UART0_TX_DEFAULT_IRQHANDLER TRUE

Enable/Disable UART0 TX Default IRQ Handler

#define CONFIG_UART0_RX_DEFAULT_IRQHANDLER TRUE

Enable/Disable UART0 RX Default IRQ Handler

#define CONFIG_UART0_TX_ENABLE_INTERRUPT TRUE

Enable/Disable(Polling) UART0 TX Interrupt

#define CONFIG_UART0_RX_ENABLE_INTERRUPT TRUE

Enable/Disable(Polling) UART0 RX Interrupt

#define CONFIG_ENABLE_DRIVER_UART1 TRUE

Enable/Disable UART Driver

#define CONFIG_UART1_TX_DEFAULT_IRQHANDLER FALSE

Enable/Disable UART1 TX Default IRQ Handler

#define CONFIG_UART1_RX_DEFAULT_IRQHANDLER FALSE

Enable/Disable UART1 RX Default IRQ Handler

#define CONFIG_UART1_TX_ENABLE_INTERRUPT FALSE

Enable/Disable(Polling) UART1 TX Interrupt

#define CONFIG_UART1_RX_ENABLE_INTERRUPT FALSE

Enable/Disable(Polling) UART1 RX Interrupt

#define CONFIG_ENABLE_DRIVER_SERIAL_FLASH TRUE

Enable/Disable Serial Flash Driver

#define CONFIG_ENABLE_DRIVER_I2C TRUE

Enable/Disable I2C Driver

#define CONFIG_I2C_DEFAULT_IRQHANDLER TRUE

Enable/Disable I2C Default IRQ Handler

#define CONFIG_I2C_ENABLE_INTERRUPT TRUE

Enable/Disable(Polling) I2C Interrupt

#define CONFIG_ENABLE_DRIVER_TIMER0 TRUE

Enable/Disable TIMER Driver

#define CONFIG_TIMER0_DEFAULT_IRQHANDLER TRUE

Enable/Disable TIMER0 Default IRQ Handler

#define CONFIG_TIMER0_ENABLE_INTERRUPT TRUE

Enable/Disable TIMER0 Interrupt

#define CONFIG_ENABLE_DRIVER_TIMER1 TRUE

Enable/Disable TIMER Driver

#define CONFIG_TIMER1_DEFAULT_IRQHANDLER TRUE

Enable/Disable TIMER1 Default IRQ Handler

#define CONFIG_TIMER1_ENABLE_INTERRUPT TRUE

Enable/Disable TIMER1 Interrupt

#define CONFIG_ENABLE_DRIVER_TIMER2 TRUE

Enable/Disable TIMER Driver

#define CONFIG_TIMER2_DEFAULT_IRQHANDLER TRUE

Enable/Disable TIMER2 Default IRQ Handler

#define CONFIG_TIMER2_ENABLE_INTERRUPT TRUE

Enable/Disable TIMER2 Interrupt

#define CONFIG_ENABLE_DRIVER_TIMER3 TRUE

Enable/Disable TIMER Driver

#define CONFIG_TIMER3_DEFAULT_IRQHANDLER TRUE

Enable/Disable TIMER3 Default IRQ Handler

#define CONFIG_TIMER3_ENABLE_INTERRUPT TRUE

Enable/Disable TIMER3 Interrupt

#define CONFIG_ENABLE_DRIVER_PWM0 TRUE

Enable/Disable PWM Driver

#define CONFIG_PWM0_DEFAULT_IRQHANDLER FALSE

Enable/Disable PWM0 Default IRQ Handler

#define CONFIG_PWM0_ENABLE_INTERRUPT FALSE

Enable/Disable PWM0 Default IRQ Handler

#define CONFIG_ENABLE_DRIVER_PWM1 TRUE

Enable/Disable PWM Driver

#define CONFIG_PWM1_DEFAULT_IRQHANDLER FALSE

Enable/Disable PWM0 Interrupt

#define CONFIG_PWM1_ENABLE_INTERRUPT FALSE

Enable/Disable PWM1 Default IRQ Handler

#define CONFIG_ENABLE_DRIVER_WDT TRUE

Enable/Disable WDT Driver

#define CONFIG_WDT_DEFAULT_IRQHANDLER TRUE

Enable/Disable WDT Default IRQ Handler

#define CONFIG_WDT_ENABLE_INTERRUPT TRUE

Enable/Disable WDT Interrupt

#define CONFIG_ENABLE_DRIVER_DMA TRUE

Enable/Disable DMA Driver

#define CONFIG_DMA_DEFAULT_IRQHANDLER TRUE

Enable/Disable DMA Default IRQ Handler

#define CONFIG_DMA_ENABLE_INTERRUPT TRUE

Enable/Disable DMA Interrupt

#define CONFIG_ENABLE_DRIVER_RTC TRUE

Enable/Disable RTC Driver

#define CONFIG_RTC_DEFAULT_IRQHANDLER TRUE

Enable/Disable RTC Default IRQ Handler

#define CONFIG_RTC_ENABLE_INTERRUPT TRUE

Enable/Disable RTC Interrupt

#define CONFIG_ENABLE_DRIVER_RTC_CAP TRUE

Enable/Disable RTC Captrue Driver

#define CONFIG_RTC_CAP_DEFAULT_IRQHANDLER TRUE

Enable/Disable RTC Captrue Default IRQ Handler

#define CONFIG_RTC_CAP_ENABLE_INTERRUPT TRUE

Enable/Disable RTC Captrue Interrupt

#define CONFIG_ENABLE_DRIVER_BLE_DP TRUE

Enable/Disable BLE datapath Driver

#define CONFIG_ENABLE_DRIVER_CALIB TRUE

Enable/Disable Calibration Driver

#define CONFIG_CALIB_DEFAULT_IRQHANDLER FALSE

Enable/Disable Calibration Default IRQ Handler

#define CONFIG_CALIB_ENABLE_INTERRUPT FALSE

Enable/Disable Calibration Interrupt

#define CONFIG_ENABLE_DRIVER_ADC TRUE

Enable/Disable ADC Driver

#define CONFIG_ADC_DEFAULT_IRQHANDLER FALSE

Enable/Disable ADC Default IRQ Handler

#define CONFIG_ADC_ENABLE_INTERRUPT FALSE

Enable/Disable ADC Interrupt

#define CONFIG_ENABLE_DRIVER_ANALOG TRUE

Enable/Disable Analog Driver

#define CONFIG_ENABLE_DRIVER_ACMP0 TRUE

Enable/Disable Analog Driver

#define CONFIG_ACMP0_DEFAULT_IRQHANDLER TRUE

Enable/Disable Analog Comparator Default IRQ Handler

#define CONFIG_ACMP0_ENABLE_INTERRUPT TRUE

Enable/Disable Analog Comparator Interrupt

#define CONFIG_ENABLE_DRIVER_ACMP1 TRUE

Enable/Disable Analog Driver

#define CONFIG_ACMP1_DEFAULT_IRQHANDLER TRUE

Enable/Disable Analog Comparator Default IRQ Handler

#define CONFIG_ACMP1_ENABLE_INTERRUPT TRUE

Enable/Disable Analog Comparator Interrupt

#define CONFIG_ENABLE_DRIVER_QNRF TRUE

Enable/Disable RF Driver

#define CONFIG_ENABLE_DRIVER_SLEEP TRUE

Enable/Disable Sleep Driver

#define CONFIG_ENABLE_DRIVER_SYSCON TRUE

Enable/Disable System Controller Driver

#define CONFIG_ENABLE_ROM_DRIVER_CALIB TRUE

Enable/Disable Calibration ROM Driver

#define GPIO_CALLBACK_EN TRUE

target configuration

Enable/Disable GPIO Driver Callback

#define UART_DMA_EN FALSE

Enable/Disable UART DMA function

#define UART_CALLBACK_EN TRUE

Enable/Disable UART Driver Callback

#define UART_BAUDRATE_TABLE_EN TRUE

Enable/Disable UART Baudrate table

#define SPI_DMA_EN FALSE

Enable/Disable SPI DMA function

#define SPI_CALLBACK_EN TRUE

Enable/Disable SPI Driver Callback

#define I2C_MODE I2C_MASTER

Config I2C Mode: Master or Slave

#define I2C_CALLBACK_EN TRUE

Enable/Disable I2C Driver Callback

#define TIMER0_CAP_MODE INCAP_EVENT_MOD

Config Timer0 Capture Mode: Input Capture timer/event/counter mode

#define TIMER1_CAP_MODE INCAP_TIMER_MOD

Config Timer1 Capture Mode: Input Capture timer/event/counter mode

#define TIMER2_CAP_MODE INCAP_EVENT_MOD

Config Timer2 Capture Mode: Input Capture timer/event/counter mode

#define TIMER3_CAP_MODE INCAP_COUNTER_MOD

Config Timer3 Capture Mode: Input Capture timer/event/counter mode

#define TIMER0_CALLBACK_EN TRUE

Enable/Disable Timer0 Driver Callback

#define TIMER1_CALLBACK_EN TRUE

Enable/Disable Timer1 Driver Callback

#define TIMER2_CALLBACK_EN TRUE

Enable/Disable Timer2 Driver Callback

#define TIMER3_CALLBACK_EN TRUE

Enable/Disable Timer3 Driver Callback

#define RTC_CALLBACK_EN TRUE

Enable/Disable RTC Driver Callback

#define RTC_CAP_CALLBACK_EN TRUE

Enable/Disable RTC Capture Driver Callback

#define USE_STD_C_LIB_TIME TRUE

Enable/Disable Standard C library function to parse date and time

#define DMA_CALLBACK_EN TRUE

Enable/Disable DMA Driver Callback

#define ADC_DMA_EN FALSE

Enable/Disable ADC DMA function

#define ADC_CALLBACK_EN TRUE

Enable/Disable ADC Driver Callback

#define ADC_WCMP_CALLBACK_EN TRUE

Enable/Disable ADC WCMP Callback

#define ACMP_CALLBACK_EN TRUE

Enable/Disable Analog Comparator Driver Callback

#define CALIB_CALLBACK_EN FALSE

Enable/Disable Calibration Driver Callback

#define SLEEP_CALLBACK_EN TRUE

Enable/Disable Sleep Wakeup Callback

#define SLEEP_CONFIG_EN TRUE

Enable/Disable User Config Before Enter Sleep

#define ACMP_WAKEUP_EN TRUE

Enable/Disable Analog comparator wakeup MCU

#define GPIO_WAKEUP_EN TRUE

Enable/Disable GPIO wakeup MCU

#define SLEEP_TIMER_WAKEUP_EN TRUE

Enable/Disable Sleep timer wakeup MCU

#define QN_LOW_POWER_MODE_EN FALSE

Enable/Disable Low power mode

#define CLOCK_32K_CORRECTION_EN FALSE

Enable/Disable 32K clock correction

#define UART_RX_ACTIVE_BIT_EN FALSE

Enable/Disable uart rx active bit set

#define SPI_RX_ACTIVE_BIT_EN FALSE

Enable/Disable spi rx active bit set

3. QN9020 BLE Profiles and Services

QN9020 SDK provides a complete package of application profiles and services that help customer to focus on application design of their products thus reduce time to market. It contains following application profiles and services:

BAS v1.0	---	Battery Service
BLP v1.0	---	Blood Pressure Profile
DIS v1.1	---	Device Information Service
FMP v1.0	---	Find Me Profile
GLP v1.0	---	Glucose Profile
HOGP v1.0	---	HID over GATT Profile
HRP v1.0	---	Heart Rate Profile
HTP v1.0	---	Health Thermometer Profile
PXP v1.0	---	Proximity Profile
SCPP v1.0	---	Scan Parameter Profile
TIP v1.0	---	Time Profile

The profiles and services support both client and server roles. Some application examples are offered to demonstrate the use of these profiles and services.

3.1 Battery Service

3.1.1 Battery Service Client API

Detailed Description

Battery Service Client APIs are used by APP to enable/disable the Battery Service Client Role, to read the value of a characteristic, or to write battery level notification configuration.

Function Documentation

void app_basc_enable_req (uint8_t *bas_nb*, struct bas_content * *bas*, uint16_t *conhdl*)

Parameters:

in	*bas_nb*	Number of BAS instances that have previously been found.
in	*bas*	Battery Service Content Structure.
in	*conhdl*	Connection handle for which the profile Locator role is enabled.

Response:

BASC_ENABLE_CFM

Description:

This API is used for enabling the Client role of the BAS. This Function contains BLE connection handle, the connection type and the previously saved and discovered BAS details on peer. The connection type may be PRF_CON_DISCOVERY (0x00) for discovery/initial configuration or PRF_CON_NORMAL (0x01) for a normal connection with a bonded device. Application shall save those information to reuse them for other connections. During normal connection, previously discovered device information can be reused.

If it is a discovery/configuration type of connection, it is useless to fill the BAS parameters (bas_nb and bas). Otherwise they will contain pertinent data which will be kept in the Client environment while enabled.

For a normal connection, the response to this request is sent right away after saving the BAS content in the environment and registering BASC in GATT to receive the notifications for the known attribute handles in BAS that would be notified (Battery Level Characteristic). For a discovery connection, discovery of the peer BAS is started and the response will be sent at the end of the discovery with the discovered attribute details.

void app_basc_rd_char_req (uint8_t *char_code*, uint8_t *bas_nb*, uint16_t *conhdl*)

Parameters:

in	*char_code*	Battery Service Characteristic Code.
in	*bas_nb*	Number of BAS instances that have previously been found.
in	*conhdl*	Connection handle for which the profile Locator role is enabled.

Response:

- BASC_BATT_LEVEL_NTF_CFG_RD_RSP
- BASC_BATT_LEVEL_PRES_FORMAT_RD_RSP
- BASC_ERROR_IND

Note:

char_code:

- BASC_RD_BAS_BATT_LEVE
- BASC_RD_BAS_BATT_LEVEL_PRES_FORMAT
- BASC_RD_BAS_BATT_LEVEL_CLI_CFG

Description:

This API shall be used to read the value of a characteristic or a descriptor in the peer device database.

void app_basc_cfg_indntf_req (uint16_t *ntf_cfg*, uint8_t *bas_nb*, uint16_t *conhdl*)

Parameters:

in	*ntf_cfg*	BAS Client configuration characteristics.
in	*bas_nb*	Number of BAS instances that have previously been found.
in	*conhdl*	Connection handle for which the profile Locator role is enabled.

Response:

BASC_WR_CHAR_RSP or BASC_ERROR_IND

Note:

ntf_cfg:

- PRF_CLI_STOP_NTFIND
- PRF_CLI_START_NTF
- PRF_CLI_START_IND

Description:

This API shall be used to enable or disable the notifications for one of the Battery Level Characteristic.

3.1.2 Battery Service Client Task API

Detailed Description

Battery Service Client Task APIs are used to handle the message from BASC or APP.

Data Structure Documentation

struct basc_enable_cfm

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	status	Status.
uint8_t	bas_nb	Number of BAS that have been found.
struct bas_content	bas	Existing handle values bas.

struct basc_error_ind

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	status	Status.

struct basc_wr_char_rsp

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	status	Status.

struct basc_batt_level_ntf_cfg_rd_rsp

Data Fields:

uint16_t	conhdl	Connection handle.
uint16_t	ntf_cfg	Notification Configuration Value.
uint8_t	bas_nb	Battery Service Instance.

struct basc_batt_level_pres_format_rd_rsp

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	bas_nb	Battery Service Instance - From 0 to BASC_NB_BAS_INSTANCES_MAX-1.
struct prf_char_pres_fmt	char_pres_format	Characteristic Presentation Format.

struct basc_batt_level_ind

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	ind_type	Indication Type.
uint8_t	batt_level	Battery Level.
uint8_t	bas_nb	Battery Service Instance - From 0 to BASC_NB_BAS_INSTANCES_MAX-1.

Function Documentation

int app_basc_enable_cfm_handler (ke_msg_id_t const *msgid*, struct basc_enable_cfm * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	BASC_ENABLE_CFM
in	*param*	Pointer to struct basc_enable_cfm
in	*dest_id*	TASK_APP
in	*src_id*	TASK_BASC

Returns:

If the message was consumed or not.

Description:

This handler is used by the Client role task to either send the discovery results of HID on the peer device and confirm enabling of the Client role, or to simply confirm enabling of Client role if it is a normal connection and the attribute details are already known.

int app_basc_error_ind_handler (ke_msg_id_t const *msgid*, struct basc_error_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	BASC_ERROR_IND
in	*param*	Pointer to struct basc_error_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_BASC

Returns:

If the message was consumed or not.

Description:

This handler is called when an error has been raised in the BASC Client role task.

int app_basc_wr_char_rsp_handler (ke_msg_id_t const *msgid*, struct basc_wr_char_rsp * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	BASC_WR_CHAR_RSP
in	*param*	Pointer to struct basc_wr_char_rsp
in	*dest_id*	TASK_APP
in	*src_id*	TASK_BASC

Returns:

If the message was consumed or not.

Description:

This handler is called when a write response has been received from the peer device after sending of a write request.

int app_basc_batt_level_ntf_cfg_rd_rsp_handler (ke_msg_id_t const *msgid*, struct basc_batt_level_ntf_cfg_rd_rsp * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	BASC_BATT_LEVEL_NTF_CFG_RD_RSP
in	*param*	Pointer to struct basc_batt_level_ntf_cfg_rd_rsp
in	*dest_id*	TASK_APP
in	*src_id*	TASK_BASC

Returns:

If the message was consumed or not.

Description:

This handler is called to inform APP about the read Client Characteristic Configuration Descriptor value for the specified Battery Level Characteristic.

int app_basc_batt_level_pres_format_rd_rsp_handler (ke_msg_id_t const *msgid*, struct basc_batt_level_pres_format_rd_rsp * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	BASC_BATT_LEVEL_PRES_FORMAT_RD_RSP
in	*param*	Pointer to struct basc_batt_level_pres_format_rd_rsp
in	*dest_id*	TASK_APP
in	*src_id*	TASK_BASC

Returns:

If the message was consumed or not.

Description:

This handler is called to inform APP about the read Characteristic Presentation Format Descriptor value for the specified Battery Level Characteristic.

int app_basc_batt_level_ind_handler (ke_msg_id_t const *msgid*, struct basc_batt_level_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	BASC_BATT_LEVEL_IND
in	*param*	Pointer to struct basc_batt_level_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_BASC

Returns:

If the message was consumed or not.

Description:

This handler is called when a Battery Level Characteristic value has been received either upon reception of a notification, or upon reception of the read response.

3.1.3 Battery 'Profile' Server

Detailed Description

The Battery Service exposes the Battery Level of a single battery or set of batteries in a device. The Battery Level may either be read, or be enabled for notification by peer device.

Function Documentation

void app_bass_create_db (uint8_t *bas_nb*, uint8_t * *features*)

Parameters:

in	*bas_nb*	Number of BAS to add
in	*features*	Indicate if battery level notify function are supported or not, possible values are: ● BAS_BATT_LVL_NTF_NOT_SUP ● BAS_BATT_LVL_NTF_SUP

Response:

BASS_CREATE_DB_CFM

Description:

Create the battery service database - at initiation.

void app_bass_enable_req (uint16_t *conhdl*, uint8_t *bas_nb*, uint8_t *sec_lvl*, uint8_t *con_type*, uint16_t * *batt_level_ntf_cfg*, uint8_t * *old_batt_lvl*, uint8_t * *current_batt_lvl*, struct prf_char_pres_fmt * *batt_level_pres_format*)

Parameters:

in	*conhdl*	Connection handle for which the battery service server is enabled
in	*bas_nb*	Number of battery service
in	*sec_lvl*	Security level required for protection of attributes. Service Hide and Disable are not permitted. Possible values are: ● PERM_RIGHT_ENABLE ● PERM_RIGHT_UNAUTH ● PERM_RIGHT_AUTH
in	*con_type*	Connection type: configuration(0) or discovery(1)

in	*batt_level_ntf_cfg*	Pointer to the Battery Level Notification Configurations
in	*old_batt_lvl*	Pointer to the Last Battery Level
in	*current_batt_lvl*	Pointer to the Current Battery Level
in	*batt_level_pres_format*	Pointer to the struct prf_char_pres_fmt containing Battery Level Characteristic Presentation Format

Response:

None

Description:

Start the battery service server - at connection.

void app_bass_batt_level_upd_req (uint16_t *conhdl*, uint8_t *bas_instance*, uint8_t *batt_level*)

Parameters:

in	*conhdl*	Connection handle for which the battery service server is enabled
in	*bas_instance*	The instances of battery service
in	*batt_level*	battery level

Response:

BASS_BATT_LEVEL_UPD_CFM

Description:

Send the battery level update - at connection.

3.1.4 Battery Service Server Task API

Detailed Description

Battery Service Service Task APIs are used to handle the message from TASK_BASS or APP.

Data Structure Documentation

struct bass_create_db_cfm

Data Fields:

uint8_t	status	Status.

struct bass_disable_ind

Data Fields:

uint16_t	conhdl	Connection Handle.
uint16_t	batt_level_ntf_cfg	Battery Level Notification configuration.
uint8_t	batt_lvl	Battery Level.

struct bass_batt_level_upd_cfm

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	status	status

struct bass_batt_level_ntf_cfg_ind

Data Fields:

uint16_t	conhdl	Connection handle.
uint16_t	ntf_cfg	Notification Configuration.
uint8_t	bas_instance	BAS instance.

Function Documentation

int app_bass_create_db_cfm_handler (ke_msg_id_t const *msgid*, struct bass_create_db_cfm * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	BASS_CREATE_DB_CFM
in	*param*	Pointer to the struct bass_create_db_cfm
in	*dest_id*	TASK_APP
in	*src_id*	TASK_BASS

Returns:

If the message was consumed or not.

Description:

This handler will be called after a database creation. The status of parameter may have the following values:

- PRF_ERR_OK
- PRF_ERR_INVALID_PARAM
- ATT_INSUFF_RESOURCE

int app_bass_disable_ind_handler (ke_msg_id_t const *msgid*, struct bass_disable_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	BASS_DISABLE_IND
in	*param*	Pointer to the struct bass_disable_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_BASS

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the Application of a correct disable.

int app_bass_error_ind_handler (ke_msg_id_t const *msgid*, struct prf_server_error_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	BASS_ERROR_IND
in	*param*	Pointer to the struct prf_server_error_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_BASS

Returns:

If the message was consumed or not.

Description:

This handler will be triggered if an error has been raised during the communication.

int app_bass_batt_level_upd_cfm_handler (ke_msg_id_t const *msgid*, struct bass_batt_level_upd_cfm * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	BASS_BATT_LEVEL_UPD_CFM
in	*param*	Pointer to the struct bass_batt_level_upd_cfm
in	*dest_id*	TASK_APP
in	*src_id*	TASK_BASS

Returns:

If the message was consumed or not.

Description:

This handler will be triggered if a notification has been sent to the peer device.

int app_bass_batt_level_ntf_cfg_ind_handler (ke_msg_id_t const *msgid*, struct bass_batt_level_ntf_cfg_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	BASS_BATT_LEVEL_NTF_CFG_IND
in	*param*	Pointer to the struct bass_batt_level_ntf_cfg_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_BASS

Returns:

If the message was consumed or not.

Description:

This handler will be triggered when the notification configuration has been modified for one of the Battery Level Characteristics.

3.2 Blood Pressure Profile

3.2.1 Blood Pressure Profile Collector API

Detailed Description

BLPC role is meant to be activated on the device that will collect the blood pressure measurements from the Blood Pressure Sensor. It implies it is a GAP Central. The FW task for this role will discover the BPS and DIS present on the peer Server, after establishing connection, and will allow configuration of the BPS attributes if so required. This file contains the implementation of this API.

Function Documentation

void app_blpc_enable_req (struct bps_content * *bps*, uint16_t *conhdl*)

Parameters:

in	*bps*	Blood Pressure Service Content Structure.
in	*conhdl*	Connection handle for which the profile blood pressure collector role is enabled.

Response:

BLPC_ENABLE_CFM

Description:

This API is used for enabling the Collector role of the Blood Pressure profile. This Function contains BLE connection handle, the connection type and the previously saved discovered BPS and DIS details on peer.

The connection type may be 0 = Connection for discovery/initial configuration or 1 = Normal connection. This parameter is used by Application to discover peer device services once at first connection. Application shall save those information to reuse them for other connections. During normal connection, previously discovered device information can be reused.

This is useful since most use cases allow Bood Pressure Sensor to disconnect the link once all measurements have been sent to Collector.

If it is a discovery /configuration type of connection, the BPS and DIS parameters are useless, they will be filled with 0's.

Otherwise they will contain pertinent data which will be kept in the Collector environment while enabled. It allows for the Application to not be aware of attribute details.

For a normal connection, the response to this request is sent right away after saving the BPS and DIS content in the environment and registering BLPC in GATT to receive the indications and notifications for the known attribute handles in BPS that would be notified/indicated. For a discovery connection, discovery of the peer BPS and DIS is started and the response will be sent at the end of the discovery with the discovered attribute details.

void app_blpc_rd_char_req (uint8_t *char_code*, uint16_t *conhdl*)

Parameters:

in	*char_code*	Code for which characteristic to read: ● BLPC_RD_BPS_FEATURE ///Read BPS Blood pressure Features ● BLPC_RD_BPS_BP_MEAS_CFG ///Read BPS Blood pressure Measurement Client Cfg. Desc ● BLPC_RD_BPS_CP_MEAS_CFG ///Read BPS Intermdiate Cuff Pressure Client Cfg. Desc
in	*conhdl*	Connection handle for which the profile blood pressure collector role is enabled.

Response:

BLPC_RD_CHAR_RSP or BLPC_ERROR_IND

Description:

This API is used by the application to send a GATT_READ_CHAR_REQ with the parameters deduced from the char_code. The definitions for the different mapping codes for characteristics that are possibly readable are in blpc.h (for BPS) and in svc.h (for DIS). Upon reception of this message, BLPC checks whether the parameters are correct, then if the handle for the characteristic is valid (not 0x0000) and the request is sent to GATT. When the peer has responded to GATT, and the response is routed to BLPC, the BLPC_RD_CHAR_RSP message will be generically built and the Application must be able to interpret it based on the read request it made. And error status is also possible either for the Read procedure or for the application request, in the second case, the BLPC_ERROR_IND message is sent to Application.

void app_blpc_cfg_indntf_req (uint8_t *char_code*, uint16_t *cfg_val*, uint16_t *conhdl*)

Parameters:

in	*char_code*	Code for which characteristic to read: ● BPS_BP_MEAS_CODE ///Blood Pressure Measurement ● BPS_INTERM_CP_CODE ///Intermediate Cuff Pressure Measurement
in	*cfg_val*	Configuration characteristics: ● PRF_CLI_STOP_NTFIND ● PRF_CLI_START_NTF ● PRF_CLI_START_IND
in	*conhdl*	Connection handle for which the profile blood pressure collector role is enabled.

Response:

BLPC_WR_CHAR_RSP or BLPC_ERROR_IND

Description:

This API is used by the application to send a GATT_WRITE_CHAR_REQ with the parameters deduced from the char_code and cfg_val. The definitions for the different codes

for characteristics that can be configured to indicate/notify are in blpc.h. Upon reception of this message, BLPC checks whether the parameters are correct, then if the handle for the characteristic is valid (not 0x0000) and the request is sent to GATT. When the peer has responded to GATT, and the response is routed to BLPC, the BLPC_WR_CHAR_RSP message will be generically built and sent to Application. An error status is also possible either for the Write procedure or for the application request, in the second case, the BLPC_ERROR_IND message is sent to Application.

3.2.2 Blood Pressure Profile Collector Task API

Detailed Description

Blood Pressure Profile Collector Task APIs are used to handle the message from BLPC or APP.

Data Structure Documentation

struct blpc_enable_cfm

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	status	status
struct bps_content	bps	Existing handle values bps.

struct blpc_error_ind

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	status	Status.

struct blpc_rd_char_rsp

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	status	Status.
struct att_info_data	data	Holder of retrieved data.

struct blpc_wr_char_rsp

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	status	Status.

struct blpc_meas_ind

Data Fields:

uint16_t	conhdl	Connection handle.
uint16_t	flag_interm_cp	Flag indicating if it is a intermediary cuff pressure measurement (1) or stable blood pressure measurement (0).
struct bps_bp_meas	meas_val	Blood Pressure measurement.

Function Documentation

int app_blpc_enable_cfm_handler (ke_msg_id_t const *msgid*, struct blpc_enable_cfm * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	BLPC_ENABLE_CFM
in	*param*	Pointer to struct blpc_enable_cfm
in	*dest_id*	TASK_APP
in	*src_id*	TASK_BLPC

Returns:

If the message was consumed or not.

Description:

This API is used by the Collector to either send the discovery results of BPS on the Blood Pressure and confirm enabling of the Collector role, or to simply confirm enabling of Collector role if it is a normal connection and the attribute details are already known.

int app_blpc_error_ind_handler (ke_msg_id_t const *msgid*, struct blpc_error_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	BLPC_ERROR_IND
in	*param*	Pointer to struct blpc_error_ind.
in	*dest_id*	TASK_APP
in	*src_id*	TASK_BLPC

Returns:

If the message was consumed or not.

Description:

This API is used by the Collector role to inform the Application of an error occurred in different situations. The error codes are proprietary and defined in prf_types.h. An error may occur during attribute discovery or due to application request parameters. Following reception of this message, the application will decide the necessary action.

int app_blpc_rd_char_rsp_handler (ke_msg_id_t const *msgid*, struct blpc_rd_char_rsp * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	BLPC_RD_CHAR_RSP
in	*param*	Pointer to struct blpc_rd_char_rsp
in	*dest_id*	TASK_APP
in	*src_id*	TASK_BLPC

Returns:

If the message was consumed or not.

Description:

This API is used by the Collector role to inform the Application of a received read response. The status and the data from the read response are passed directly to Application, which must interpret them based on the request it made.

int app_blpc_wr_char_rsp_handler (ke_msg_id_t const *msgid*, struct blpc_wr_char_rsp * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	BLPC_WR_CHAR_RSP
in	*param*	Pointer to struct blpc_wr_char_rsp
in	*dest_id*	TASK_APP
in	*src_id*	TASK_BLPC

Returns:

If the message was consumed or not.

Description:

This API is used by the Collector role to inform the Application of a received write response. The status and the data from the write response are passed directly to Application, which must interpret them based on the request it made.

int app_blpc_bp_meas_ind_handler (ke_msg_id_t const *msgid*, struct blpc_meas_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	BLPC_BP_MEAS_IND
in	*param*	Pointer to struct blpc_meas_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_BLPC

Returns:

If the message was consumed or not.

Description:

This API is used by the Collector role to inform the Application of a received blood pressure value, either by notification (flag_interm_cp = intermediate) or indication (flag_interm_cp = stable). The application will do what it needs to do with the received measurement. No confirmation of reception is needed because the GATT sends it directly to the peer.

3.2.3 Blood Pressure Profile Sensor

Detailed Description

Blood Pressure Profile Sensor (BLPS): A BLPS (e.g. PC, phone, etc) is the term used by this profile to describe a device that can perform blood pressure measurement and notify about on-going measurement and indicate final result to a peer BLE device.

Function Documentation

void app_blps_create_db (uint8_t *features*)

Parameters:

in	*features*	Blood Pressure features used to create database. Possible values are: BLPS_INTM_CUFF_PRESS_SUP

Response:

BLPS_CREATE_DB_CFM

Description:

This function shall be called after system power-on (or after GAP Reset) in order to create blood pressure profile database. This database will be visible from a peer device bun not usable until BLPS_ENABLE_REQ message is sent within a BLE connection

Note:

The Blood Pressure profile requires the presence of three DIS characteristics : Manufacturer Name String, Model Number, System Identifier. It is application's responsibility to add an instance of the DIS into the database by using the Device information create database API.

void app_blps_enable_req (uint16_t *conhdl*, uint8_t *sec_lvl*, uint8_t *con_type*, uint16_t *bp_meas_ind_en*, uint16_t *interm_cp_ntf_en*, uint16_t *bp_feature*)

Parameters:

in	*conhdl*	Connection handle for which the profile blood pressure sensor role is enabled.
in	*sec_lvl*	Security level required for protection of IAS attributes, Service Hide and Disable are not permitted. Possible values are:

		• PERM_RIGHT_ENABLE • PERM_RIGHT_UNAUTH • PERM_RIGHT_AUTH
in	*con_type*	Connection type: configuration(0) or discovery(1) Normal connection: Peer device is known and client configuration characteristics shall be restored. Discovery connection: Peer device is unknown and peer collector will manage client configuration characteristics.
in	*bp_meas_ind_en*	Value stored for Blood Pressure indication Client Configuration Char.
in	*interm_cp_ntf_en*	Value stored for intermediate cuff pressure notification Client Configuration Char.
in	*bp_feature*	Specific blood pressure feature description, Possible values are(See blp_common.h): • BPS_F_BODY_MVMT_DETECT_NOT_SUPPORTED • BPS_F_BODY_MVMT_DETECT_SUPPORTED • BPS_F_CUFF_FIT_DETECT_NOT_SUPPORTED • BPS_F_CUFF_FIT_DETECT_SUPPORTED • BPS_F_IRREGULAR_PULSE_DETECT_NOT_SUPPORTED • BPS_F_IRREGULAR_PULSE_DETECT_SUPPORTED • BPS_F_PULSE_RATE_RANGE_DETECT_NOT_SUPPORTED • BPS_F_PULSE_RATE_RANGE_DETECT_SUPPORTED • BPS_F_MEAS_POS_DETECT_NOT_SUPPORTED • BPS_F_MEAS_POS_DETECT_SUPPORTED • BPS_F_MULTIPLE_BONDS_NOT_SUPPORTED • BPS_F_MULTIPLE_BONDS_SUPPORTED

Response:

None

Description:

This function is used for enabling the Blood Pressure Sensor role of the Blood Pressure profile. Before calling this function, a BLE connection shall exist with peer device.

void app_blps_pressure_send_req (uint16_t *conhdl*, uint8_t *flag_interm*, struct bps_bp_meas * *meas_val*)

Parameters:

in	*conhdl*	Connection handle for which the profile blood pressure sensor role is enabled
in	*flag_interm*	Own code for differentiating between Blood Pressure Measurement, and Intermediate Cuff Pressure Measurement characteristics
in	*meas_val*	Pointer to the struct bps_bp_meas containing Blood Pressure measurement value

Response:

BLPS_MEAS_SEND_CFM or None

Description:

This function is used by the application (which handles the blood pressure device driver and measurements) to send a blood pressure measurement through the blood pressure sensor role.

Note:

Message BLPS_CFG_INDNTF_IND will be received as a hint to call this function

3.2.4 Blood Pressure Profile Sensor Task API

Detailed Description

Blood Pressure Profile Sensor Task APIs are used to handle the message from BLPS or APP.

Data Structure Documentation

struct blps_create_db_cfm

Data Fields:

uint8_t	status	Status.

struct blps_disable_ind

Data Fields:

uint16_t	conhdl	
uint16_t	bp_meas_ind_en	Blood Pressure indication configuration.
uint16_t	interm_cp_ntf_en	Intermediate Cuff Pressure Notification configuration.

struct blps_cfg_indntf_ind

Data Fields:

uint16_t	conhdl	Connection handle.
uint16_t	cfg_val	Stop/notify/indicate value to configure into the peer characteristic.
uint8_t	char_code	Own code for differentiating between Blood Pressure Measurement, and Intermediate Cuff Pressure Measurement characteristics

struct blps_meas_send_cfm

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	status	Status.

Function Documentation

int app_blps_create_db_cfm_handler (ke_msg_id_t const *msgid*, struct blps_create_db_cfm * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	BLPS_CREATE_DB_CFM
in	*param*	Pointer to the struct blps_create_db_cfm
in	*dest_id*	TASK_APP
in	*src_id*	TASK_BLPS

Returns:

If the message was consumed or not.

Description:

This handler will be called after a database creation. It contains status of database creation.

int app_blps_disable_ind_handler (ke_msg_id_t const *msgid*, struct blps_disable_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	BLPS_DISABLE_IND
in	*param*	Pointer to the struct blps_disable_ind
in	*dest_id*	TASK_APP

in	*src_id*	TASK_BLPS

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the Application of a correct disable. The configuration that the collector has set in BPS attributes must be conserved and the 4 values that are important are sent back to the application for safe keeping until the next time this profile role is enabled.

int app_blps_cfg_indntf_ind_handler (ke_msg_id_t const *msgid*, struct blps_cfg_indntf_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	BLPS_CFG_INDNTF_IND
in	*param*	Pointer to the struct blps_cfg_indntf_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_BLPS

Returns:

If the message was consumed or not.

Description:

This handler is used to inform application that peer device has changed notification configuration.

int app_blps_meas_send_cfm_handler (ke_msg_id_t const *msgid*, struct blps_meas_send_cfm * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	BLPS_MEAS_SEND_CFM
in	*param*	Pointer to the struct blps_meas_send_cfm
in	*dest_id*	TASK_APP
in	*src_id*	TASK_BLPS

Returns:

If the message was consumed or not.

Description:

This handler is used to inform to the application a confirmation, or error status of a notification request being sent to GATT for the Intermediate Cuff Pressure Char.

int app_blps_error_ind_handler (ke_msg_id_t const *msgid*, struct prf_server_error_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	BLPS_ERROR_IND
in	*param*	Pointer to the struct prf_server_error_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_BLPS

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the Application of an occurred error information

3.3 Device Information Service

3.3.1 Device Information Service Client API

Detailed Description

DISC role is meant to be activated on the device that will locate the Server. It implies it is a GAP Central. The FW task for this role will discover the Device Information Service present on the peer Server, after establishing connection, and will allow reading different information about the device. This file contains the implementation of this API.

Function Documentation

void app_disc_enable_req (struct disc_dis_content * *dis*, uint16_t *conhdl*)

Parameters:

in	*dis*	Device Information Service Content Structure.
in	*conhdl*	Connection handle for which the profile Locator role is enabled.

Response:

DISC_ENABLE_CFM

Description:

This API is used for enabling the Client role of the Device Information Service. The Application sends it, and it contains the connection handle for the connection this profile is activated, the connection type and the previously saved discovered DIS details on peer.

The connection type may be 0 = Connection for discovery or 1 = Normal connection. This difference has been made and Application would handle it in order to not discover the DIS on the Server at every connection, but do it only once and keep the discovered details in the Client device between connections.

If it is a discovery type connection, the DIS parameter is useless, and it will be filled with 0's. Otherwise it will contain pertinent data which will be kept in the Client environment while enabled. It allows for the Application to not be aware of attribute details. For a normal connection, the response to this request is sent right away after saving the dis content in the environment. For a discovery connection, discovery of the peer DIS is started and the response will be sent at the end of the discovery with the discovered attribute details.

void app_disc_rd_char_req (uint8_t *char_code*, uint16_t *conhdl*)

Parameters:

in	*char_code*	Code of the characteristic.
in	*conhdl*	Connection handle for which the profile Locator role is enabled.

Response:

DISC_RD_CHAR_RSP or DISC_ERROR_IND

Note:

char_code:

- DISC_MANUFACTURER_NAME_CHAR
- DISC_MODEL_NB_STR_CHAR
- DISC_SERIAL_NB_STR_CHAR
- DISC_HARD_REV_STR_CHAR
- DISC_FIRM_REV_STR_CHAR
- DISC_SW_REV_STR_CHAR
- DISC_SYSTEM_ID_CHAR
- DISC_IEEE_CHAR
- DISC_PNP_ID_CHAR

Description:

This API is used by the application to send a GATT_READ_CHAR_REQ with the parameters deduced from the char_code. Upon reception of this message, DISC checks

whether the parameters are correct, then if the handle for the characteristic is valid (not 0x0000) and the request is sent to GATT.

When the peer has responded to GATT, and the response is routed to DISC, the DICS_RD_CHAR_RSP message will be generically built and the Application must be able to interpret it based on the read request it made. And error status is also possible either for the Read procedure or for the application request, in the second case, the DISC_ERROR_IND message is sent to Application.

No parsing intelligence of the received response is added in this API handler, so all the work of interpretation must be added in the Application depending of its request and use of the response.

3.3.2 Device Information Service Client Task API

Detailed Description

Device Information Service Client Task APIs are used to handle the message from DISC or APP

Data Structure Documentation

struct disc_enable_cfm

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	status	status
struct disc_dis_content	dis	DIS handle values and characteristic properties.

struct disc_rd_char_rsp

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	status	Status.
uint8_t	char_code	Char. Code.
uint16_t	val_length	Value Length.
uint8_t	val	Value.

Function Documentation

int app_disc_enable_cfm_handler (ke_msg_id_t const *msgid*, struct disc_enable_cfm * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	DISC_ENABLE_CFM
in	*param*	Pointer to struct disc_enable_cfm
in	*dest_id*	TASK_APP
in	*src_id*	TASK_DISC

Returns:

If the message was consumed or not.

Description:

This API is used by the Client to either send the discovery results of DIS on Server and confirm enabling of the Client role, or to simply confirm enabling of Client role if it is a

normal connection and the DIS details are already known. An error may have also occurred and is signaled.

int app_disc_rd_char_rsp_handler (ke_msg_id_t const *msgid*, struct disc_rd_char_rsp * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	DISC_RD_CHAR_RSP
in	*param*	Pointer to struct disc_rd_char_rsp
in	*dest_id*	TASK_APP
in	*src_id*	TASK_DISC

Returns:

If the message was consumed or not.

Description:

This API is used by the Client role to inform the Application of a received read response. The status and the data from the read response are passed directly to Application, which must interpret them based on the request it made.

int app_disc_disable_ind_handler (ke_msg_id_t const *msgid*, struct prf_client_disable_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	DISC_DISABLE_IND
in	*param*	Pointer to struct prf_client_disable_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_DISC

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the application that the Device Information Client Role task has been correctly disabled or if an error has occurred during this process.

3.3.3 Device Information Service Server

Detailed Description

The Bluetooth Low Energy Device Information Service enables the user to expose manufacturer and/or vendor information about a device.

Function Documentation

void app_diss_create_db (uint16_t *features*)

Parameters:

in	*features*	Indicate characteristics that are supported, possible values are • DIS_MANUFACTURER_NAME_CHAR_SUP • DIS_MODEL_NB_STR_CHAR_SUP • DIS_SERIAL_NB_STR_CHAR_SUP • DIS_HARD_REV_STR_CHAR_SUP • DIS_FIRM_REV_STR_CHAR_SUP • DIS_SW_REV_STR_CHAR_SUP • DIS_SYSTEM_ID_CHAR_SUP • DIS_IEEE_CHAR_SUP • DIS_PNP_ID_CHAR_SUP

Response:

DISS_CREATE_DB_CFM

Description:

This function shall be used to add an instance of the Device Information Service into the database. This should be done during the initialization phase of the device.

Note:

All characteristics of the Device Information Service are optional. However, some profiles require the presence of several of these characteristics. Please refer to the specification of these profiles for more information.

void app_diss_enable_req (uint16_t *conhdl*, uint8_t *sec_lvl*)

Parameters:

in	*conhdl*	Connection handle for which the profile Target role is enabled
in	*sec_lvl*	Security level required for protection of attributes, Service Hide and Disable are not permitted. Possible values are: ● PERM_RIGHT_ENABLE ● PERM_RIGHT_UNAUTH ● PERM_RIGHT_AUTH

Response:

None

Description:

This function is used for enabling the Server role of the device information service

void app_set_char_val_req (uint8_t *char_code*, uint8_t *val_len*, uint8_t * *val*)

Parameters:

in	*char_code*	Characteristic Code, possible values are: ● DIS_MANUFACTURER_NAME_CHAR ● DIS_MODEL_NB_STR_CHAR ● DIS_SERIAL_NB_STR_CHAR ● DIS_HARD_REV_STR_CHAR ● DIS_FIRM_REV_STR_CHAR ● DIS_SW_REV_STR_CHAR ● DIS_SYSTEM_ID_CHAR ● DIS_IEEE_CHAR ● DIS_PNP_ID_CHAR
in	*val_len*	Value length
in	*val*	Pointer to value

Response:

None

Description:

This function is used to initialize any of the characteristic values before a connection with a peer device.

3.3.4 Device Information Service Server Task API

Detailed Description

Device Information Service Server Task APIs are used to handle the message from DISS or APP

Data Structure Documentation

struct diss_create_db_cfm

Data Fields:

uint8_t	status	Status.

struct diss_disable_ind

Data Fields:

uint16_t	conhdl	Connection handle.

Function Documentation

int app_diss_create_db_cfm_handler (ke_msg_id_t const *msgid*, struct diss_create_db_cfm * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	DISS_CREATE_DB_CFM
in	*param*	Pointer to the struct diss_create_db_cfm
in	*dest_id*	TASK_APP
in	*src_id*	TASK_DISS

Returns:

If the message was consumed or not.

Description:

This handler will be called after a database creation. The status parameter indicates if the DIS has been successfully added or not. Possible values for the status are: ATT_ERR_NO_ERROR and ATT_INSUFF_RESOURCE.

int app_diss_error_ind_handler (ke_msg_id_t const *msgid*, struct prf_server_error_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	DISS_ERROR_IND
in	*param*	Pointer to the struct prf_server_error_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_DISS

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the Application of an occurred error.

3.4 Fine Me Profile

3.4.1 Find Me Locator API

Detailed Description

FINDL role is meant to be activated on the device that will locate the Target. It implies it is a GAP Central. The FW task for this role will discover the Immediate Alert Service present on the peer Server, after establishing connection, and will allow writing different alert levels to the Alert Level characteristic in the IAS. This file contains the implementation of this API

Function Documentation

void app_findl_enable_req (struct ias_content * *ias*, uint16_t *conhdl*)

Parameters:

in	*ias*	IAS details.

in	*conhdl*	Connection handle for which the profile Locator role is enabled.

Response:

FINDL_ENABLE_CFM

Description:

This API is used for enabling the Locator role of the Find Me profile. This function contains the connection handle for the connection this profile is activated, the connection type and the previously saved discovered IAS details on peer.

The connection type may be 0 = Connection for discovery or 1 = Normal connection. This difference has been made and Application would handle it in order to not discover the IAS on the Target at every connection, but do it only once and keep the discovered details in the Locator device between connections. ATTENTION: Normally information about the peer should not be kept from one connection to the next if they have not bonded!

If it is a discovery type connection, the IAS parameter is useless, and it will be filled with 0's. Otherwise it will contain pertinent data which will be kept in the Locator environment while enabled. It allows for the Application to not be aware of attribute details. For a normal connection, the response to this request is sent right away after saving the ias content in the environment. For a discovery connection, discovery of the peer IAS is started and the response will be sent at the end of the discovery with the discovered attribute details.

void app_findl_set_alert_req (uint8_t *alert_lvl*, uint16_t *conhdl*)

Parameters:

in	*alert_lvl*	Alert level.
in	*conhdl*	Connection handle for which the profile Locator role is enabled.

Response:

None

Note:

Alert level:

- FINDL_ALERT_NONE
- FINDL_ALERT_MILD
- FINDL_ALERT_HIGH

Description:

This API is used by the application to trigger/stop and alert on the peer Target device. The Locator role environment contains the attribute handle for the Alert Level Characteristic in the IAS of the Target peer device. This way, a correct request to write this attribute to the level requested by the application can be sent. Since the Alert Level Characteristic in IAS can only be written using a Write No Response ATT Request, no confirmation can be received through the profile. The only confirmation can be observed by the user either by noticing and alarm on the Target device or the alarm stopping.

3.4.2 Find Me Locator Task API

Detailed Description

Find Me Locator Task APIs are used to handle the message from FINDL or APP.

Data Structure Documentation

struct findl_error_ind

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	status	Status.

struct findl_enable_cfm

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	status	Status.
struct ias_content	ias	IAs attribute content.

Function Documentation

int app_findl_enable_cfm_handler (ke_msg_id_t const *msgid*, struct findl_enable_cfm * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	FINDL_ENABLE_CFM
in	*param*	Pointer to struct findl_enable_cfm
in	*dest_id*	TASK_APP
in	*src_id*	TASK_FINDL

Returns:

If the message was consumed or not.

Description:

This API is used by the Locator to either send the discovery results of IAS on Target and confirm enabling of the Locator role, or to simply confirm enabling of Locator role if it is a normal connection and the IAS details are already known. An error may have also occurred and is signaled.

int app_findl_error_ind_handler (ke_msg_id_t const *msgid*, struct findl_error_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	FINDL_ERROR_IND
in	*param*	Pointer to struct findl_error_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_FINDL

Returns:

If the message was consumed or not.

Description:

This API is used by the application to trigger/stop and alert on the peer Target device. The Locator role environment contains the attribute handle for the Alert Level Characteristic in the IAS of the Target peer device. This way, a correct request to write this attribute to the level requested by the application can be sent. Since the Alert Level Characteristic in IAS can only be written using a Write No Response ATT Request, no confirmation can be received through the profile. The only confirmation can be observed by the user either by noticing and alarm on the Target device or the alarm stopping.

int app_findl_disable_ind_handler (ke_msg_id_t const *msgid*, struct prf_client_disable_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	FINDL_DISABLE_IND
in	*param*	Pointer to struct prf_client_disable_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_FINDL

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the application that the Find Me Profile Locator Role task has been correctly disabled or if an error has occurred during this process.

3.4.3 Find Me Target

Detailed Description

The Find Me profile defines the behavior when a button is pressed on a device to cause an immediate alert on a peer device. This can be used to allow users to find devices that have been misplaced. Within the profile, two roles can be supported: Locator and Target. The Find Me Target shall be a server. The Find Me Locator shall be a client. When the Find Me Locator device wishes to cause an alert on the Find Me Target device, it shall write the specific Alert Level in the Alert Level characteristic.

Function Documentation

void app_findt_create_db (void)

Response:

FINDT_CREATE_DB_CFM

Description:

This functiuon shall be used to add an instance of the Immediate Alert Service into the database. This should be done during the initialization phase of the device. The status parameter indicates if the IAS has been successfully added or not. Possible values for the status are: ATT_ERR_NO_ERROR and ATT_INSUFF_RESOURCE.

void app_findt_enable_req (uint16_t *conhdl*, uint8_t *sec_lvl*)

Parameters:

in	*conhdl*	Connection handle for which the profile Target role is enabled
in	*sec_lvl*	Security level required for protection of attributes Service Hide and Disable are not permitted. Possible values are: ● PERM_RIGHT_ENABLE ● PERM_RIGHT_UNAUTH ● PERM_RIGHT_AUTH

Response:

None

Description:

This functiuon is used for enabling the Target role of the Find Me profile. It contains the connection handle for the connection this profile is activated.

3.4.4 Find Me Profile Target Task API

Detailed Description

Find Me Profile Target Task APIs are used to handle the message from FINDT or APP.

Data Structure Documentation

struct findt_create_db_cfm

Data Fields:

uint8_t	status	Status.

struct findt_disable_ind

Data Fields:

uint16_t	conhdl	Connection handle.

struct findt_alert_ind

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	alert_lvl	Alert level.

Function Documentation

int app_findt_create_db_cfm_handler (ke_msg_id_t const *msgid*, struct findt_create_db_cfm * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	FINDT_CREATE_DB_CFM
in	*param*	Pointer to the struct findt_create_db_cfm
in	*dest_id*	TASK_APP
in	*src_id*	TASK_FINDT

Returns:

If the message was consumed or not.

Description:

This handler will be called after a database creation. The status parameter indicates if the IAS has been successfully added or not. Possible values for the status are: ATT_ERR_NO_ERROR and ATT_INSUFF_RESOURCE.

int app_findt_alert_ind_handler (ke_msg_id_t const *msgid*, struct findt_alert_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	FINDT_ALERT_IND
in	*param*	Pointer to the struct findt_alert_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_FINDT

Returns:

If the message was consumed or not.

Description:

This handler is calld to inform the Application of a valid alert level written by the peer in the IAS Alert Level Characteristic. Possible values are: No Alert(0), Mild Alert(1), High Alert(2) The Application alone is responsible for actually triggering/stopping a noticeable visual/audio alert on the device upon reception of message FINDT_ALERT_IND.

int app_findt_disable_ind_handler (ke_msg_id_t const *msgid*, struct findt_disable_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	FINDT_DISABLE_IND
in	*param*	Pointer to the struct findt_disable_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_FINDT

Returns:

If the message was consumed or not.

Description:

This handler is calld to inform the application of a correct disable. It will be triggered after a disconnection with the peer device.

int app_findt_error_ind_handler (ke_msg_id_t const *msgid*, struct prf_server_error_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	FINDT_ERROR_IND
in	*param*	Pointer to the struct prf_server_error_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_FINDT

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the application of an occurred error information

3.5 Glucose Profile

3.5.1 Glucose Profile Collector API

Detailed Description

GLPC role is meant to be activated on the device that will collect the Glucose measurements from the Glucose sensor. It implies it is a GAP Central. The FW task for this role will discover the GLS present on the peer Server, after establishing connection, and will allow configuration of the GLS attributes if so required.

Function Documentation

void app_glpc_enable_req (struct gls_content * *gls*, uint16_t *conhdl*)

Parameters:

in	*gls*	Existing handle values GLS (see Glucose Content Structure (struct gls_content))
in	*conhdl*	Connection handle for which the profile Glucose collector role is enabled

Response:

GLPC_ENABLE_CFM

Description:

This API is used for enabling the Collector role of the Glucose profile. This function contains BLE connection handle, the connection type and the previously saved discovered GLS details on peer. The connection type may be 0 = Connection for discovery/initial configuration or 1 = Normal connection. This parameter is used by Application to discover peer device services once at first connection. Application shall save those information to reuse them for other connections. During normal connection, previously discovered device information can be reused.

This is useful since most use cases allow Glucose sensor to disconnect the link once all measurements have been sent to Collector.

If it is a discovery /configuration type of connection, the GLS parameters are useless, and they will be filled with 0's. Otherwise they will contain pertinent data which will be kept in the Collector environment while enabled. It allows for the Application to not be aware of attribute details.

For a normal connection, the response to this request is sent right away after saving the GLS content in the environment and registering GLPC in GATT to receive the indications and notifications for the known attribute handles in GLS that would be notified/indicated. For a discovery connection, discovery of the peer GLS is started and the response will be sent at the end of the discovery with the discovered attribute details.

void app_glpc_register_req (bool *meas_ctx_en*, uint16_t *conhdl*)

Parameters:

in	*meas_ctx_en*	Register or not Glucose measurement context notifications
in	*conhdl*	Connection handle for which the profile Glucose collector role is enabled

Response:

GLPC_REGISTER_CFM

Description:

This API is used by the application to register to Glucose sensor notifications and indications. According to peer available characteristics, it performs in one action all event registration. This shall be performed after enabling collector first time Glucose sensor is used. This registration shall be kept by peer device if bonding procedure has been performed.

This procedure shall be done before doing any Record Access Control Point requests.

void app_glpc_read_features_req (uint16_t *conhdl*)

Parameters:

in	*conhdl*	Connection handle for which the profile Glucose collector role is enabled

Response:

GLPC_READ_FEATURES_RSP

Description:

This API is used by the application to read peer Glucose sensor features.

void app_glpc_racp_req (struct glp_racp_req * *racp*, uint16_t *conhdl*)

Parameters:

in	*racp*	Record Access Control Point (RACP) Request (struct glp_racp_req)
in	*conhdl*	Connection handle for which the profile Glucose collector role is enabled

Response:

GLPC_RACP_RSP

Description:

This API is used by Application to request execution of a RACP Request on peer Glucose sensor. This action could be report glucose measurements, report number of measurement, delete measurements or abort an on-going operation (see Record Access Control Point (RACP) OP Code). This action contains a filter describing which glucose measurement are concerned by the operation.

Possible OP Code:

- GLP_REQ_REP_STRD_RECS: Report stored records
- GLP_REQ_REP_NUM_OF_STRD_RECS: Report number of stored records
- GLP_REQ_DEL_STRD_RECS: Delete stored records
- GLP_REQ_ABORT_OP: Abort on-going operation.

Possible Operator:

- GLP_OP_ALL_RECS
- GLP_OP_LT_OR_EQ
- GLP_OP_GT_OR_EQ

- GLP_OP_WITHIN_RANGE_OF
- GLP_OP_FIRST_REC
- GLP_OP_LAST_REC

Possible filter type:

- GLP_FILTER_SEQ_NUMBER
- GLP_FILTER_USER_FACING_TIME

Note:

During an on-going operation, any other request from collector shall be refused by Glucose service, except GLP_REQ_ABORT_OP (Abort operation). In that case on-going operation shall be stopped by glucose sensor. RACP response message shall be received from peer Glucose sensor with GLP_REQ_ABORT_OP op_code and status equals GLP_RSP_SUCCESS.

3.5.2 Glucose Profile Collector Task API

Detailed Description

Glucose Profile Collector Task APIs are used to handle the message from GLPC or APP.

Data Structure Documentation

struct glpc_enable_cfm

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	status	status
struct gls_content	gls	Existing handle values gls.

struct glpc_register_cfm

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	status	Status.

struct glpc_read_features_rsp

Data Fields:

uint16_t	conhdl	Connection handle.
uint16_t	features	Glucose sensor features.
uint8_t	status	Status.

struct glpc_racp_rsp

Data Fields:

uint16_t	conhdl	Connection handle.
struct glp_racp_rsp	racp_rsp	record access control response
uint8_t	status	Status.

struct glpc_meas_ind

Data Fields:

uint16_t	conhdl	Connection handle.
uint16_t	seq_num	Sequence Number.
struct glp_meas	meas_val	Glucose measurement.

struct glpc_meas_ctx_ind

Data Fields:

uint16_t	conhdl	Connection handle.
uint16_t	seq_num	Sequence Number.
struct glp_meas_ctx	ctx	Glucose measurement.

Function Documentation

int app_glpc_enable_cfm_handler (ke_msg_id_t const *msgid*, struct glpc_enable_cfm * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GLPC_ENABLE_CFM
in	*param*	Pointer to the struct glpc_enable_cfm
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GLPC

Returns:

If the message was consumed or not.

Description:

This API is used by the Collector to either send the discovery results of GLS on the Glucose sensor and confirm enabling of the Collector role, or to simply confirm enabling of Collector role if it is a normal connection and the attribute details are already known.

int app_glpc_register_cfm_handler (ke_msg_id_t const *msgid*, struct glpc_register_cfm * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GLPC_REGISTER_CFM
in	*param*	Pointer to the struct glpc_register_cfm
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GLPC

Returns:

If the message was consumed or not.

Description:

This API is used by the Collector role to inform the Application about Glucose sensor event registration status.

int app_glpc_read_features_rsp_handler (ke_msg_id_t const *msgid*, struct glpc_read_features_rsp * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GLPC_READ_FEATURES_RSP
in	*param*	Pointer to the struct glpc_read_features_rsp
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GLPC

Returns:

If the message was consumed or not.

Description:

This API is used by the Collector role to inform the Application of received peer Glucose sensor features.

int app_glpc_racp_response_handler (ke_msg_id_t const *msgid*, struct glpc_racp_rsp * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GLPC_RACP_RSP
in	*param*	Pointer to the struct glpc_racp_rsp

in	*dest_id*	TASK_APP
in	*src_id*	TASK_GLPC

Returns:

If the message was consumed or not.

Description:

This API is used by the Collector role to inform the Application of a status of Record Access Control Point Action. It shall contain status of executed request or number of stored measurement records if GLP_REQ_REP_NUM_OF_STRD_RECS has been requested.

int app_glpc_meas_ind_handler (ke_msg_id_t const *msgid*, struct glpc_meas_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GLPC_MEAS_IND
in	*param*	Pointer to the struct glpc_meas_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GLPC

Returns:

If the message was consumed or not.

Description:

This API is used by the Collector role to inform the Application of a received Glucose measurement value. This value should be received within a RACP request (GLP_REQ_REP_STRD_RECS), but it could be send out of request by Glucose sensor.

int app_glpc_meas_ctx_ind_handler (ke_msg_id_t const *msgid*, struct glpc_meas_ctx_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GLPC_MEAS_CTX_IND
in	*param*	Pointer to the struct glpc_meas_ctx_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GLPC

Returns:

If the message was consumed or not.

Description:

This API is used by the Collector role to inform the Application of a received Glucose measurement context value. This value should be received within a RACP request (GLP_REQ_REP_STRD_RECS), but it could be send out of request by Glucose sensor. It shall be trigger by Glucose sensor only if corresponding glucose measurement previously received has GLP_MEAS_CTX_INF_FOLW in its measurement flag.

3.5.3 Glucose Profile Sensor

Detailed Description

The Bluetooth Low Energy Glucose profile enables the user to manage measurements from a Glucose sensor device and also configure it for different use cases. Within the profile, two roles can be supported: Collector and Sensor. The Glucose Sensor shall be a Server. The Collector shall be a Client.

The functionality of the profile requires the presence of certain services and attributes on one of the two devices, which the other device can manipulate. In this case, the Glucose device must have one instance of the Glucose Service (GLS) and one instance of Device Information Service(DIS) in its attribute database. The Glucose Profile Collector (GLPC) will discover these services and their characteristics, and it may then configure them to cause the Glucose Profile Sensor (GLPS) device to take measurements and notify them to the Collector.

Function Documentation

void app_glps_create_db (uint16_t *start_hdl*, uint8_t *meas_ctx_supported*)

Parameters:

in	*start_hdl*	Glucose Service start handle.Set it to 0 for automatic handle allocation
in	*meas_ctx_supported*	Flag used to add or not measurement context in database

Response:

GLPS_CREATE_DB_CFM

Description:

This function shall be called after system power-on (or after GAP Reset) in order to create Glucose profile database. This database will be visible from a peer device but not usable until glps enabled within a BLE connection.

void app_glps_enable_req (uint16_t *conhdl*, uint16_t *features*, uint8_t *sec_lvl*, uint8_t *con_type*, uint16_t *evt_cfg*)

Parameters:

in	*conhdl*	Connection handle for which the profile Glucose sensor role is enabled
in	*features*	Glucose sensor features(see enum glp_srv_feature_flag in glp_common.h)
in	*sec_lvl*	Security level required for protection of GLS attributes: Service Hide and Disable are not permitted. Possible values are: ● PERM_RIGHT_ENABLE ● PERM_RIGHT_UNAUTH ● PERM_RIGHT_AUTH
in	*con_type*	Connection type: configuration(0) or discovery(1)
in	*evt_cfg*	Glucose sensor event configuration (notification, indication) configured by peer device during another connection (Bonded information) ● bit 1: Glucose measurement notifications enabled ● bit 2: Glucose measurement context notifications enabled ● bit 4: Record Access Control Point (RACP) indications enabled

Response:

GLPS_ENABLE_CFM

Description:

This function is used for enabling the Glucose Sensor role. Before calling this function, a BLE connection shall exist with peer device. Application shall provide connection handle in order to activate the profile.

void app_glps_racp_rsp_req_send (uint16_t *conhdl*, uint16_t *num_of_record*, uint8_t *op_code*, uint8_t *status*)

Parameters:

in	*conhdl*	Connection handle for which the profile Glucose sensor role is enabled
in	*num_of_record*	Number of records found(Should be set only if RACP operation code equals GLP_REQ_REP_NUM_OF_STRD_RECS)
in	*op_code*	RACP Request operation code(see enum glp_racp_op_code in file

		glp_common.h)
in	*status*	RACP Request operation status code(see enum glp_racp_status in file glp_common.h)

Response:

GLPS_REQ_CMP_EVT

Description:

This function is used by the application to send Record Access Control Point (RACP) request response. If requested operation is GLP_REQ_REP_NUM_OF_STRD_RECS, number of stored record should be set, else it will be ignored by Glucose sensor role. Status code should be set according to Glucose profile error code (see enum glp_racp_status in file glp_common.h)

void app_glps_meas_without_ctx_req_send (uint16_t *conhdl*, uint16_t *seq_num*, struct glp_meas * *meas*)

Parameters:

in	*conhdl*	Connection handle for which the profile Glucose sensor role is enabled
in	*seq_num*	Measurement Sequence Number
in	*meas*	Pointer to the struct glp_meas containing Glucose measurement

Response:

GLPS_REQ_CMP_EVT

Description:

This function is used by the application (which handles the Glucose device driver and measurements) to send a glucose measurement without following measurement context information.

void app_glps_meas_with_ctx_req_send (uint16_t *conhdl*, uint16_t *seq_num*, struct glp_meas * *meas*, struct glp_meas_ctx * *ctx*)

Parameters:

in	*conhdl*	Connection handle for which the profile Glucose sensor role is enabled
in	*seq_num*	Measurement sequence number
in	*meas*	Pointer to the struct glp_meas containing Glucose measurement
in	*ctx*	Pointer to the struct glp_meas_ctx containing Glucose measurement context structure

Response:

GLPS_REQ_CMP_EVT

Description:

This function is used by the application (which handles the Glucose device driver and measurements) to send a glucose measurement with following measurement context information.

3.5.4 Glucose Profile Sensor Task API

Detailed Description

Glucose Profile Sensor Task APIs are used to handle the message from GLPS or APP.

Data Structure Documentation

struct glps_create_db_cfm

Data Fields:

uint8_t	status	Status.

struct glps_enable_cfm

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	status	Status.

struct glps_disable_ind

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	evt_cfg	Glucose indication/notification configuration.

struct glps_cfg_indntf_ind

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	evt_cfg	Glucose indication/notification configuration.

struct glps_racp_req_ind

Data Fields:

uint16_t	conhdl	Connection handle.
struct glp_racp_req	racp_req	RACP Request.

struct glps_req_cmp_evt

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	request	completed request
uint8_t	status	Command status.

Function Documentation

int app_glps_create_db_cfm_handler (ke_msg_id_t const *msgid*, struct glps_create_db_cfm * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GLPS_CREATE_DB_CFM
in	*param*	Pointer to the struct glps_create_db_cfm
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GLPS

Returns:

If the message was consumed or not.

Description:

This handler will be called after a database creation. It contains status of database creation.

int app_glps_enable_cfm_handler (ke_msg_id_t const *msgid*, struct glps_enable_cfm * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GLPS_ENABLE_CFM
in	*param*	Pointer to the struct glps_enable_cfm
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GLPS

Returns:

If the message was consumed or not.

Description:

This handler is used inform the Application that it has been enabled or not.

int app_glps_disable_ind_handler (ke_msg_id_t const *msgid*, struct glps_disable_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GLPS_DISABLE_IND
in	*param*	Pointer to the struct glps_disable_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GLPS

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the Application of a correct disable or inform that a disconnection happened (information in status).

int app_glps_cfg_indntf_ind_handler (ke_msg_id_t const *msgid*, struct glps_cfg_indntf_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GLPS_CFG_INDNTF_IND
in	*param*	Pointer to the struct glps_cfg_indntf_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GLPS

Returns:

If the message was consumed or not.

Description:

This handler is triggered when peer device modify notification/indication configuration of Glucose Sensor role characteristics. If peer device has been bonded, configuration that collector has set in GLS attributes (evt_cfg) shall be kept by application in a non-volatile memory for next time this profile role is enabled.

Note:

Glucose sensor event configuration (notification, indication) configured by peer device (Bonded information)

- bit 1: Glucose measurement notifications enabled
- bit 2: Glucose measurement context notifications enabled
- bit 4: Record Access Control Point (RACP) indications enabled

int app_glps_racp_req_ind_handler (ke_msg_id_t const *msgid*, struct glps_racp_req_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GLPS_RACP_REQ_IND
in	*param*	Pointer to the struct glps_racp_req_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GLPS

Returns:

If the message was consumed or not.

Description:

This handler is triggered when peer collector request to perform a Record Access Control Point (RACP) action. This action could be report glucose measurements, report number of measurement, delete measurements or abort an on-going operation (see Record Access Control Point (RACP) OP Code). This action contains a filter describing which glucose measurement are concerned by the operation. Possible operations:

- GLP_REQ_REP_STRD_RECS: Report stored records

- GLP_REQ_REP_NUM_OF_STRD_RECS: Report number of stored records
- GLP_REQ_DEL_STRD_RECS: Delete stored records
- GLP_REQ_ABORT_OP: Abort on-going operation

Note:

During an on-going operation, any other request from peer device will be automatically refused by Glucose service, except GLP_REQ_ABORT_OP (Abort operation). In that case on-going operation shall be stopped. Finally application shall send response with GLP_REQ_ABORT_OP op_code and status equals GLP_RSP_SUCCESS.

int app_glps_req_cmp_evt_handler (ke_msg_id_t const *msgid*, struct glps_req_cmp_evt * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GLPS_REQ_CMP_EVT
in	*param*	Pointer to the struct glps_req_cmp_evt
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GLPS

Returns:

If the message was consumed or not.

Description:

This handler is triggered when a requested action has been performed: GLPS_SEND_MEAS_REQ_NTF_CMP: Glucose measurement notification sent completed GLPS_SEND_RACP_RSP_IND_CMP: Record Access Control Point Response Indication sent completed

int app_glps_error_ind_handler (ke_msg_id_t const *msgid*, struct prf_server_error_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GLPS_ERROR_IND
in	*param*	Pointer to the struct prf_server_error_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GLPS

Returns:

If the message was consumed or not.

Description:

This handler is used to report an occurred error.

3.6 HID over GATT Profile

3.6.1 HID Over GATT Profile Boot Host Role API

Detailed Description

The BLE HOGP Boot Host role has been designed to allow a collector to easily commuicate with a HID Boot device (Boot Keyboard or Boot Mouse). All data exchanged within this role have a fixed length and each bit in a packet has a known meaning. There is no need of a Report descriptor, so no HID Parser is required in the application. The table below shown Boot Host characteristic requirements:

Report Map: Excluded
Report: Excluded
Boot Keyboard Input/Output Report: Mandatory to support at least one of these features
Boot Mouse Input Report: Mandatory to support at least one of these features
HID Information: Excluded
HID Control Point: Excluded

Protocol Mode: Mandatory

Thus, the HOGP Boot Host role task will only look for the non-excluded chracteristics during the discovery process (more details in HOGPBH_ENABLE_REQ and HOGPBH_ENABLE_CFM). Some restrictions have been defined in BLE HOGP specification and shall be respected by the application designer:

A Boot Host shall not concurrently be a Report Host.
The Boot Host shall use the GAP Central role.

As we currently have a static implementation, some firmware limitations have been defined for this role (in the hogpbh.h file):

The maximal number of HID Service instances that can be handled has been limited to 2 (HOGPBH_NB_HIDS_INST_MAX).

Function Documentation

void app_hogpbh_enable_req (uint8_t *hids_nb*, struct hids_content * *hids*, uint16_t *conhdl*)

Parameters:

in	*hids_nb*	Number of instances of the HID Service that have been found during the last discovery
in	*hids*	Information about HID Services that have been found during the last discovery
in	*conhdl*	Connection handle

Response:

HOGPBH_ENABLE_CFM

Description:

This API is used for enabling the Boot Host role of the HOGP. This function contains BLE connection handle, the connection type and the previously saved discovered HIDS details on peer.

The connection type may be PRF_CON_DISCOVERY (0x00) for discovery/initial configuration or PRF_CON_NORMAL (0x01) for a normal connection with a bonded device. Application shall save those information to reuse them for other connections. During normal connection, previously discovered device information can be reused.

If it is a discovery/configuration type of connection, it is useless to fill the HIDS parameters (hids_nb and hids) are useless. Otherwise they will contain pertinent data which will be kept in the Boot Host environment while enabled.

For a normal connection, the response to this request is sent right away after saving the HIDS content in the environment and registering HOGPBH in GATT to receive the notifications for the known attribute handles in HIDS that would be notified (Boot Keyboard Input Report and Boot Mouse Input Report). For a discovery connection, discovery of the peer HIDS is started and the response will be sent at the end of the discovery with the discovered attribute details.

void app_hogpbh_disable_req (uint16_t *conhdl*)

Parameters:

in	*conhdl*	Connection handle

Response:

None

Description:

This API is used for disabling the Boot Host role of the HOGP. The function contains the connection handle for the connection this profile is activated.

void app_hogpbh_rd_char_req (uint8_t *char_code*, uint8_t *hids_nb*, uint16_t *conhdl*)

Parameters:

in	*char_code*	Characteristic value code: ● HOGPBH_RD_HIDS_PROTO_MODE ● HOGPBH_RD_HIDS_BOOT_KB_IN_REPORT ● HOGPBH_RD_HIDS_BOOT_KB_OUT_REPORT ● HOGPBH_RD_HIDS_BOOT_MOUSE_IN_REPORT ● HOGPBH_RD_HIDS_BOOT_KB_IN_REPORT_CFG ● HOGPBH_RD_HIDS_BOOT_MOUSE_IN_REPORT_CFG
in	*hids_nb*	HIDS instance
in	*conhdl*	Connection handle

Response:

The response depends on the read_code parameter value. If an error has been raised before this message is sent in the air, a HOGPBH_ERR_IND message is sent; if the received value doesn't match with requirements or implementation limitations a HOGPBH_RD_CHAR_ERR_RSP is sent.

Description:

This API shall be used to read the value of a characteristic or a descriptor in the HID Device database.

When the HOGP Boot Host task receives this message, the handler checks several parameters. If one of these don't match requirements, a HOGPBH_ERROR_IND is sent to the application. The table below resumes the possible status values:

- PRF_ERR_INVALID_PARAM (0x81): Either the provided Connection Handle is unknown, or the specified HIDS instance is upper than the number of found HIDS.
- PRF_ERR_INEXISTENT_HDL (0x82): The required attribute has not been found in the peer device database.

void app_hogpbh_cfg_ntf_req (uint8_t *desc_code*, uint16_t *ntf_cfg*, uint8_t *hids_nb*, uint16_t *conhdl*)

Parameters:

in	*desc_code*	Client Characteristic Configuration Descriptor Code :

- HOGPBH_DESC_BOOT_KB_IN_REPORT_CFG,
- HOGPBH_DESC_BOOT_MOUSE_IN_REPORT_CFG,

in	*ntf_cfg*	Configuration value to write:

- PRF_CLI_STOP_NTFIND
- PRF_CLI_START_NTF
- PRF_CLI_START_IND

in	*hids_nb*	HIDS instance
in	*conhdl*	Connection handle

Response:

HOGPBH_WR_CHAR_RSP or HOGPBH_ERROR_IND

Description:

This API shall be used to enable or disable the notifications for either the Boot Keyboard Input Characteristic or the Boot Mouse Input Characteristic.

When the HOGP Boot Host task receives this message, the handler checks several parameters. If one of these don't match requirements, a HOGPBH_ERROR_IND is sent to the application. The table below resumes the possible status values:

- PRF_ERR_INVALID_PARAM (0x81): Either the provided Connection Handle is unknown, or the specified HIDS instance is upper than the number of found HIDS, or the ntf_cfg parameter value is not valid.
- PRF_ERR_INEXISTENT_HDL (0x82): The required attribute has not been found in the peer device database.

void app_hogpbh_boot_report_wr_req (uint8_t *wr_type*, uint8_t *char_code*, uint8_t *report_length*, uint8_t *hids_nb*, uint8_t * *report*, uint16_t *conhdl*)

Parameters:

in	*wr_type*	Write type (Write or Write without Response)
in	*char_code*	char code
in	*report_length*	Report data length
in	*hids_nb*	HIDS instance
in	*report*	Boot Keyboard Output Report Characteristic value to write
in	*conhdl*	Connection handle

Response:

None or HOGPRH_WR_CHAR_RSP

Description:

This API shall be used to write the value of a Boot Keyboard Output Report Characteristic in the peer device database.

void app_hogpbh_set_boot_proto_mode_req (uint8_t *hids_nb*, uint16_t *conhdl*)

Parameters:

in	*hids_nb*	HIDS instance
in	*conhdl*	Connection handle

Response:

HOGPRH_WR_CHAR_RSP or HOGPRH_ERROR_IND

Description:

This API shall be used to set the protocol mode of a HID Service instance to the Boot Procotol Mode.

The default protocol mode for a HID Device able to support either the Boot protocol mode or the Report protocol mode (the Protocol Mode characteristic shall be present in its database) is the Report protocol mode. Thus, the application shall send this message right after the end of the discovery.

When the HOGP Boot Host task receives this message, the handler checks several parameters. If one of these don't match requirements, a HOGPBH_ERROR_IND is sent to the application. The table below resumes the possible status values:

- PRF_ERR_INVALID_PARAM 0x81 Either the provided Connection Handle is unknown, or the specified HIDS instance is upper than the number of found HIDS.
- PRF_ERR_INEXISTENT_HDL 0x82 The required attribute has not been found in the peer device database

3.6.2 HID Over GATT Profile Boot Host Role TASK

Detailed Description

HID Over GATT Profile Boot Host Role TASK APIs are used to handle the message from HOGPBH or APP.

Data Structure Documentation

struct hogpbh_enable_cfm

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	status	status
uint8_t	hids_nb	Number of HIDS instances.
struct hids_content	hids	Existing handle values hids.

struct hogpbh_cfg_ntf_rd_rsp

Data Fields:

uint16_t	conhdl	Connection handle.
uint16_t	cfg_val	Stop/notify value to configure into the peer characteristic.
uint8_t	desc_code	Client Characteristic Configuration Code.
uint8_t	hids_nb	HIDS instance.

struct hogpbh_char_req_rsp

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	status	Status.
uint8_t	att_code	Attribute Code.

struct hogpbh_proto_mode_rd_rsp

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	proto_mode	Protocol Mode.
uint8_t	hids_nb	HIDS Instance.

struct hogpbh_boot_report_ind

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	hids_nb	HIDS Instance.
uint8_t	ind_type	Read Response or Notification.
uint8_t	char_code	Char Code.
uint8_t	report_length	Report Length.
uint8_t	report	Boot Report.

Function Documentation

int app_hogpbh_enable_cfm_handler (ke_msg_id_t const *msgid*, struct hogpbh_enable_cfm * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HOGPBH_ENABLE_CFM
in	*param*	Pointer to the struct hogpbh_enable_cfm
in	*dest_id*	TASK_APP
in	*src_id*	TASK_HOGPBH

Returns:

If the message was consumed or not.

Description:

This API is used by the Boot Host to either send the discovery results of HIDS on the HID device and confirm enabling of the Boot Host role, or to simply confirm enabling of Boot Host role if it is a normal connection and the attribute details are already known.

int app_hogpbh_wr_char_rsp_handler (ke_msg_id_t const *msgid*, struct hogpbh_char_req_rsp * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HOGPBH_WR_CHAR_RSP
in	*param*	Pointer to the struct hogpbh_char_req_rsp
in	*dest_id*	TASK_APP
in	*src_id*	TASK_HOGPBH

Returns:

If the message was consumed or not.

Description:

The API is used to inform the application about the status of the writing request that has been sent.

int app_hogpbh_cfg_ntf_rd_rsp_handler (ke_msg_id_t const *msgid*, struct hogpbh_cfg_ntf_rd_rsp * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HOGPBH_CFG_NTF_RD_RSP
in	*param*	Pointer to the struct hogpbh_cfg_ntf_rd_rsp
in	*dest_id*	TASK_APP
in	*src_id*	TASK_HOGPBH

Returns:

If the message was consumed or not.

Description:

This API is used to inform the application about the read Client Characteristic Configuration Descriptor value.

int app_hogpbh_proto_mode_rd_rsp_handler (ke_msg_id_t const *msgid*, struct hogpbh_proto_mode_rd_rsp * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HOGPBH_PROTO_MODE_RD_RSP
in	*param*	Pointer to the struct hogpbh_proto_mode_rd_rsp
in	*dest_id*	TASK_APP
in	*src_id*	TASK_HOGPBH

Returns:

If the message was consumed or not.

Description:

This API is used to inform the application about the read Protocol Mode Characteristic value.

int app_hogpbh_boot_report_ind_handler (ke_msg_id_t const *msgid*, struct hogpbh_boot_report_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HOGPBH_BOOT_REPORT_IND
in	*param*	Pointer to the struct hogpbh_boot_report_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_HOGPBH

Returns:

If the message was consumed or not.

Description:

This API is used to inform the application about the read Boot Keyboard Input Report Characteristic value.

int app_hogpbh_char_req_rsp_handler (ke_msg_id_t const *msgid*, struct hogpbh_char_req_rsp * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HOGPBH_RD_CHAR_ERR_RSP
in	*param*	Pointer to the struct hogpbh_char_req_rsp
in	*dest_id*	TASK_APP
in	*src_id*	TASK_HOGPBH

Returns:

If the message was consumed or not.

Description:

This handler is called when application sent a read request is not complient with the specification or the implementation limitations.

3.6.3 HID Over GATT Profile device

Function Documentation

void app_hogpd_create_db (uint8_t *hids_nb*, struct hogpd_hids_cfg * *cfg*)

Parameters:

in	*hids_nb*	Number of HID Service instances to add in the database
in	*cfg*	Pointer to the struct hogpd_hids_cfg containing Configuration for each HID Service you want to add

Response:

HOGPD_CREATE_DB_CFM

Description:

This function shall be used to add one or more instance of the HID Service in the database.

Note:

Multiples service instances of the HID Service should allow implementers to define HID Devices whose combined functions require more than 512 octets of data to describe. Thus, the second instance of the HID Service shall exist only if the Report Characteristic value exceeds 512 bytes.

void app_hogpd_report_map_req (uint16_t *report_map_len*, uint8_t *hids_nb*, uint8_t * *report_map*)

Parameters:

in	*report_map_len*	Length of the Report Map Characteristic value
in	*hids_nb*	HID Service instance the Report Map Characteristic belongs to
in	*report_map*	Pointer to the Report Map Characteristic value

Response:

None or HOGPD_ERROR_IND

Description:

This function shall be used to initialize the Report Map Characteristic value in the database. This value is not supposed to change during the connection or during the device life cycle. According to the BLE HOGP specification, the Report Characteristic value length is limited to 512 bytes. If the value has been set with success in the database, no response is sent to the application. If an error is raised, a HOGPD_ERROR_IND message will be sent with one the following error status: PRF_ERR_INVALID_PARAM The specified Report Map Characteristic value length is upper than the limitation(512 bytes by default) PRF_ERR_REQ_DISALLOWED The required HID Service has not been added in the database

void app_hogpd_enable_req (uint16_t *conhdl*, uint8_t *sec_lvl*, uint8_t *con_type*, uint8_t *hids_nb*, struct hogpd_hids_ntf_cfg * *ntf_cfg*)

Parameters:

in	*conhdl*	Connection handle
in	*sec_lvl*	Required security level. Service Hide and Disable are not permitted.

		Possible values are: ● PERM_RIGHT_ENABLE ● PERM_RIGHT_UNAUTH ● PERM_RIGHT_AUTH
in	*con_type*	Connection type
in	*hids_nb*	Number of HID Service instances
in	*ntf_cfg*	Saved notification configurations

Response:

None or HOGPD_ERROR_IND

Description:

This function shall be used after the connection with a peer device has been established in order to enable the HOGP device role task for the specified connection.

void app_hogpd_report_upd_req (uint16_t *conhdl*, uint8_t *hids_nb*, uint8_t *report_nb*, uint8_t *report_length*, uint8_t * *report*)

Parameters:

in	*conhdl*	Connection handle
in	*hids_nb*	HID Service instance
in	*report_nb*	Report Characteristic instance
in	*report_length*	Length of the Report Characteristic value
in	*report*	Report Characteristic value

Response:

HOGPD_NTF_SENT_CFM

Description:

This function is used to update the value of the Report Characteristic stored in the database and to notify the Host about this new value if sending of notifications has been enabled for it.

void app_hogpd_boot_report_upd_req (uint16_t *conhdl*, uint8_t *hids_nb*, uint8_t *char_code*, uint8_t *report_length*, uint8_t * *boot_report*)

Parameters:

in	*conhdl*	Connection handle
in	*hids_nb*	HID Service instance
in	*char_code*	Characteristic code is used to indicate keyboard or mouse data, possible values are: ● HOGPD_BOOT_KB_IN_REPORT_CHAR ● HOGPD_BOOT_MOUSE_IN_REPORT_CHAR ● HOGPD_BOOT_KB_OUT_REPORT_CHAR
in	*report_length*	Report data length
in	*boot_report*	Boot Report Characteristic value

Response:

HOGPD_NTF_SENT_CFM

Description:

This function is used to update the value of the Boot Report Characteristic value stored in the database and to notify the Host about this new value if sending of notifications has been enabled for it.

3.6.4 HID Over GATT Device Task API

Detailed Description

HID Over GATT Device Task APIs are used to handle the message from HOGPD or APP.

Data Structure Documentation

struct hogpd_create_db_cfm

Data Fields:

uint8_t	status	Status.

struct hogpd_disable_ind

Data Fields:

uint16_t	conhdl	Connection handle.
struct hogpd_hids_ntf_cfg	ntf_cfg	Notification Configurations.

struct hogpd_ntf_cfg_ind

Data Fields:

uint16_t	conhdl	Connection Handle.
uint16_t	ntf_en	New Notification Configuration Value.
uint8_t	cfg_code	Cfg. Code.
uint8_t	hids_nb	HIDS Instance.
uint8_t	report_nb	Report Char. Instance.

struct hogpd_proto_mode_ind

Data Fields:

uint16_t	conhdl	Connection Handle.
uint8_t	hids_nb	HIDS Instance.
uint8_t	proto_mode	New Protocol Mode Characteristic Value.

struct hogpd_report_ind

Data Fields:

uint16_t	conhdl	Connection Handle.
uint8_t	hids_nb	HIDS Instance.
uint8_t	report_nb	Report Char. Instance.
uint8_t	report_length	Report Length.
uint8_t	report	Report.

struct hogpd_boot_kb_in_ind

Data Fields:

uint16_t	conhdl	Connection Handle.
uint8_t	hids_nb	HIDS Instance.
uint8_t	boot_kb_in_report	Boot Keyboard Input Report.

struct hogpd_boot_mouse_in_ind

Data Fields:

uint16_t	conhdl	Connection Handle.
uint8_t	hids_nb	HIDS Instance.
uint8_t	report_len	Boot Mouse Input Report Length.
uint8_t	boot_mouse_in_report	Boot Mouse Input Report.

struct hogpd_boot_kb_out_ind

Data Fields:

uint16_t	conhdl	Connection Handle.
uint8_t	hids_nb	HIDS Instance.
uint8_t	boot_kb_out_report	Boot Keyboard Output Report.

struct hogpd_ctnl_pt_ind

Data Fields:

uint16_t	conhdl	Connection Handle.
uint8_t	hids_nb	HIDS Instance.
uint8_t	hid_ctnl_pt	New HID Control Point Characteristic Value.

struct hogpd_ntf_sent_cfm

Data Fields:

uint16_t	conhdl	Connection Handle.
uint8_t	status	Status.
uint8_t	hids_nb	HIDS Instance.
uint8_t	char_code	Characteristic Code.
uint8_t	report_nb	Report Instance.

Function Documentation

int app_hogpd_create_db_cfm_handler (ke_msg_id_t const *msgid*, struct hogpd_create_db_cfm * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HOGPD_CREATE_DB_CFM
in	*param*	Pointer to the struct hogpd_create_db_cfm
in	*dest_id*	TASK_APP
in	*src_id*	TASK_HOGPD

Returns:

If the message was consumed or not.

Description:

This handler will be called after a database creation. It contains status of database creation.

int app_hogpd_disable_ind_handler (ke_msg_id_t const *msgid*, struct hogpd_disable_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HOGPD_DISABLE_IND
in	*param*	Pointer to the struct hogpd_disable_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_HOGPD

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the Application of a correct disable.

int app_hogpd_error_ind_handler (ke_msg_id_t const *msgid*, struct prf_server_error_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HOGPD_ERROR_IND
in	*param*	Pointer to the struct prf_server_error_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_HOGPD

Returns:

If the message was consumed or not.

Description:

This handler will be called if an error has been raised during the communication.

int app_hogpd_ntf_cfg_ind_handler (ke_msg_id_t const *msgid*, struct hogpd_ntf_cfg_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HOGPD_NTF_CFG_IND
in	*param*	Pointer to the struct hogpd_ntf_cfg_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_HOGPD

Returns:

If the message was consumed or not.

Description:

None

int app_hogpd_proto_mode_ind_handler (ke_msg_id_t const *msgid*, struct hogpd_proto_mode_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HOGPD_PROTO_MODE_IND
in	*param*	Pointer to the struct hogpd_proto_mode_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_HOGPD

Returns:

If the message was consumed or not.

Description:

This handler will be called if a Protocol Mode characteristic value has been written by a peer device.

int app_hogpd_report_ind_handler (ke_msg_id_t const *msgid*, struct hogpd_report_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HOGPD_REPORT_IND
in	*param*	Pointer to the struct hogpd_report_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_HOGPD

Returns:

If the message was consumed or not.

Description:

This handler will be called after the peer Host has written the value of one of the Report Characteristics.

int app_hogpd_boot_kb_in_ind_handler (ke_msg_id_t const *msgid*, struct hogpd_boot_kb_in_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HOGPD_BOOT_KB_IN_IND
in	*param*	Pointer to the struct hogpd_boot_kb_in_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_HOGPD

Returns:

If the message was consumed or not.

Description:

This handler will be called after the peer Host has written the value of the Boot Keyboard Input Report Characteristic.

int app_hogpd_boot_mouse_in_ind_handler (ke_msg_id_t const *msgid*, struct hogpd_boot_mouse_in_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HOGPD_BOOT_MOUSE_IN_IND
in	*param*	Pointer to the struct hogpd_boot_mouse_in_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_HOGPD

Returns:

If the message was consumed or not.

Description:

This handler will be called after the peer Host has written the value of the Boot Mouse Input Report Characteristic.

int app_hogpd_boot_kb_out_ind_handler (ke_msg_id_t const *msgid*, struct hogpd_boot_kb_out_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HOGPD_BOOT_KB_OUT_IND
in	*param*	Pointer to the struct hogpd_boot_kb_out_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_HOGPD

Returns:

If the message was consumed or not.

Description:

This handler will be called after the peer Host has written the value of the Boot Keyboard Output Report Characteristic.

int app_hogpd_ctnl_pt_ind_handler (ke_msg_id_t const *msgid*, struct hogpd_ctnl_pt_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HOGPD_CTNL_PT_IND
in	*param*	Pointer to the struct hogpd_ctnl_pt_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_HOGPD

Returns:

If the message was consumed or not.

Description:

This handler will be called each time the host enables or disables sending of notifications for characteristic.

int app_hogpd_ntf_sent_cfm_handler (ke_msg_id_t const *msgid*, struct hogpd_ntf_sent_cfm * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HOGPD_NTF_SENT_CFM
in	*param*	Pointer to the struct hogpd_ntf_sent_cfm
in	*dest_id*	TASK_APP
in	*src_id*	TASK_HOGPD

Returns:

If the message was consumed or not.

Description:

This handler will be called after reception of the HOGPD_REPORT_UPD_REQ or the HOGPD_BOOT_KB_IN_UPD_REQ or the HOGPD_BOOT_MOUSE_IN_UPD_REQ message to inform it if a notificatio has been sent to the Host or if an error has been raised. The following status code may be handled by application

- PRF_ERR_OK
- PRF_ERR_INVALID_PARAM
- PRF_ERR_REQ_DISALLOWED
- PRF_ERR_NTF_DISABLED
- PRF_ERR_FEATURE_NOT_SUPPORTED

3.6.5 HID Over GATT Profile Report Host Role API

Detailed Description

The BLE HOGP Report Host role has been designed to allow a collector to communicate with a HID device. An application which would use this role shall support a HID Parser and be able to handle arbitrary format for data transfers. The table below shown Report Host characteristic requirements:

Report Map: Mandatory
Report: Mandatory
Boot Keyboard Input Report: Excluded
Boot Keyboard Output Report: Excluded
Boot Mouse Input Report: Excluded
HID Information: Mandatory
HID Control Point: Mandatory if the Host supports Suspend Mode, otherwise optional.
Protocol Mode: Optional

Thus, the HOGP Report Host role task will only look for the non-excluded chracteristics during the discovery process. Some restrictions have been defined in the BLE HOGP specification and shall be respected by the application designer:

A Report Host shall not concurrently be a Boot Host.
The Report Host shall use the GAP Central role.

As we currently have a static implementation, some firmware limitations have been defined for this role (in the hogprh.h file):

The maximal number of HID Service instances that can be handled has been limited to 2 (HOGPBH_NB_HIDS_INST_MAX).
The maximal number of Report Characteristics instances that can be handled has been limited to 5 (HOGPRH_NB_REPORT_INST_MAX).
The maximal length of a Report Characteristic value has been limited to 45 bytes.

Function Documentation

void app_hogprh_enable_req (uint8_t *hids_nb*, struct hogprh_hids_content * *hids*, uint16_t *conhdl*)

Parameters:

in	*hids_nb*	Number of instances of the HID Service that have been found during the last discovery.
in	*hids*	Information about the HID Service instances that have been found during the last discovery.
in	*conhdl*	Connection handle

Response:

HOGPRH_ENABLE_CFM

Description:

This API is used for enabling the Report Host role of the HOGP. This function contains BLE connection handle, the connection type and the previously saved discovered HIDS details on peer.

The connection type may be PRF_CON_DISCOVERY (0x00) for discovery/initial configuration or PRF_CON_NORMAL (0x01) for a normal connection with a bonded device. Application shall save those information to reuse them for other connections. During normal connection, previously discovered device information can be reused.

If it is a discovery /configuration type of connection, it is useless to fill the HIDS parameters (hids_nb and hids) are useless. Otherwise they will contain pertinent data which will be kept in the Boot Host environment while enabled.

For a normal connection, the response to this request is sent right away after saving the HIDS content in the environment and registering HOGPRH in GATT to receive the notifications for the known attribute handles in HIDS that would be notified (Report Characteristic). For a discovery connection, discovery of the peer HIDS is started and the response will be sent at the end of the discovery with the discovered attribute details.

void app_hogprh_disable_req (uint16_t *conhdl*)

Parameters:

in	*conhdl*	Connection handle

Response:

None

Description:

This API is used for disabling the Report Host role of the HOGP. The Application sends it, and it contains the connection handle for the connection this profile is activated.

void app_hogprh_rd_char_req (uint8_t *read_code*, uint8_t *report_nb*, uint8_t *hids_nb*, uint16_t *conhdl*)

Parameters:

in	*read_code*	Characteristic Code: ● HOGPRH_RD_HIDS_REPORT_MAP ● HOGPRH_RD_HIDS_HID_INFO ● HOGPRH_RD_HIDS_PROTOCOL_MODE ● HOGPRH_RD_HIDS_REPORT ● HOGPRH_RD_HIDS_REPORT_MAP_EXT_REF_REF ● HOGPRH_RD_HIDS_REPORT_REF ● HOGPRH_RD_HIDS_REPORT_CFG
in	*report_nb*	Report Characteristic instance if needed
in	*hids_nb*	HID Service instance
in	*conhdl*	Connection handle

Response:

The response depends on the read_code parameter value. If an error has been raised before this message is sent in the air, a HOGPRH_ERR_IND message is sent; if the received value doesn't match with requirements or implementation limitations a HOGPRH_RD_CHAR_ERR_RSP is sent.

Description:

This API shall be used to read the value of a characteristic or a descriptor in the HID Device database.

When the HOGP Report Host task receives this message, the handler checks several parameters. If one of these don't match requirements, a HOGPRH_ERROR_IND is sent to the application. The table below resumes the possible status values:

- PRF_ERR_INVALID_PARAM (0x81): Either the provided Connection Handle is unknown, or the specified HIDS instance is upper than the number of found HIDS, or the specified Report Characteristic instance is upper than the limitation.

- PRF_ERR_INEXISTENT_HDL (0x82): The required attribute has not been found in the peer device database.

void app_hogprh_cfg_ntf_req (uint8_t *report_nb*, uint16_t *ntf_cfg*, uint8_t *hids_nb*, uint16_t *conhdl*)

Parameters:

in	*report_nb*	Report Characteristic instance.
in	*ntf_cfg*	Configuration value to write: ● PRF_CLI_STOP_NTFIND ● PRF_CLI_START_NTF ● PRF_CLI_START_IND
in	*hids_nb*	HID Service instance
in	*conhdl*	Connection handle

Response:

HOGPRH_WR_CHAR_RSP or HOGPRH_ERROR_IND

Description:

This API shall be used to enable or disable the notifications for a Report Characteristic instance.

When the HOGP Report Host task receives this message, the handler checks several parameters. If one of these don't match requirements, a HOGPRH_ERROR_IND is sent to the application. The table below resumes the possible status values:

- PRF_ERR_INVALID_PARAM (0x81): Either the provided Connection Handle is unknown, or the specified HIDS instance is upper than the number of found HIDS, or the specified Report Characteristic instance is upper than the limitation or the ntf_cfg parameter value is not valid.
- PRF_ERR_INEXISTENT_HDL (0x82): The required attribute has not been found in the peer device database. Either the Report Characteristic instance doesn't exist in the peer device database or this characteristic doesn't support notification (not an Input Report).

void app_hogprh_hid_ctnl_pt_wr_req (uint8_t *ctnl_pt*, uint8_t *hids_nb*, uint16_t *conhdl*)

Parameters:

in	*ctnl_pt*	HID Control Point value: ● HOGP_CTNL_PT_SUSPEND ● HOGP_CTNL_PT_EXIT_SUSPEND
in	*hids_nb*	HID Service instance
in	*conhdl*	Connection handle

Response:

None or HOGPRH_ERROR_IND

Description:

This API shall be used to write the HID Control Point Characteristic value in the peer device database.

When the HOGP Report Host task receives this message, the handler checks several parameters. If one of these don't match requirements, a HOGPRH_ERROR_IND is sent to the application. The table below resumes the possible status values:

- PRF_ERR_INVALID_PARAM (0x81): Either the provided Connection Handle is unknown, or the specified HIDS instance is upper than the number of found HIDS or the hid_ctnl_pt parameter value is not valid.
- PRF_ERR_INEXISTENT_HDL (0x82): The required attribute has not been found in the peer device database. The HID Control Point Characteristic is mandatory. Thus, this error should never happened if the discovery process has been successful.

void app_hogprh_set_report_proto_mode_req (uint8_t *hids_nb*, uint16_t *conhdl*)

Parameters:

in	*hids_nb*	HID Service instance
in	*conhdl*	Connection handle

Response:

None or HOGPRH_ERROR_IND

Description:

This API shall be used to set the protocol mode of a HID Service instance to the Report Procotol Mode.

When the HOGP Report Host task receives this message, the handler checks several parameters. If one of these don't match requirements, a HOGPRH_ERROR_IND is sent to the application. The table below resumes the possible status values:

- PRF_ERR_INVALID_PARAM (0x81): Either the provided Connection Handle is unknown, or the specified HIDS instance is upper than the number of found HIDS.
  PRF_ERR_INEXISTENT_HDL (0x82): The required attribute has not been found in the peer device database.

void app_hogprh_report_wr_req (uint8_t *report_nb*, uint8_t *report_length*, uint8_t *out_report_type*, uint8_t * *report*, uint8_t *hids_nb*, uint16_t *conhdl*)

Parameters:

in	*report_nb*	Report Characteristic instance
in	*report_length*	Length of the Report value to write
in	*out_report_type*	Type of write to perform if the Report is an Output Report
in	*report*	Report Characteristic value
in	*hids_nb*	HID Service instance
in	*conhdl*	Connection handle

Response:

None or HOGPRH_WR_CHAR_RSP or HOGPRH_ERROR_IND

Description:

This API shall be used to write the value of a Report Characteristic in the peer device database.

3.6.6 HID Over GATT Profile Report Host Role TASK

Detailed Description

HID Over GATT Profile Report Host Role TASK APIs are used to handle the message from HOGPRH or APP.

Data Structure Documentation

struct hogprh_enable_cfm

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	status	status
uint8_t	hids_nb	Number of HIDS instances.
struct hogprh_hids_content	hids	Existing handle values hids.

struct hogprh_char_req_rsp

Data Fields:

uint16_t	conhdl	Connection handle.

uint8_t	status	Status.
uint8_t	att_code	Attribute Code.

struct hogprh_proto_mode_rd_rsp

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	proto_mode	Protocol Mode.
uint8_t	hids_nb	HIDS Instance.

struct hogprh_report_map_rd_rsp

Data Fields:

uint16_t	conhdl	Connection handle.
uint16_t	report_map_length	Report Map Length.
uint8_t	hids_nb	HIDS Instance.
uint8_t	report_map	Report Map value.

struct hogprh_report_map_ext_rep_ref_rd_rsp

Data Fields:

uint16_t	conhdl	Connection handle.
uint16_t	ext_report_ref	Report Map Char. External Report Reference Descriptor value.
uint8_t	hids_nb	HIDS Instance.

struct hogprh_report_ref_rd_rsp

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	hids_nb	HIDS Instance.
uint8_t	report_nb	Report Char. Instance.
struct hids_report_ref	report_ref	Report Char. Report Reference Descriptor value.

struct hogprh_hid_info_rd_rsp

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	hids_nb	HIDS Instance.
struct hids_hid_info	hid_info	HID Information value.

struct hogprh_cfg_ntf_rd_rsp

Data Fields:

uint16_t	conhdl	Connection handle.
uint16_t	cfg_val	Stop/notify value to configure into the peer characteristic.
uint8_t	report_nb	Report instance.
uint8_t	hids_nb	HIDS instance.

struct hogprh_report_ind

Data Fields:

uint16_t	conhdl	Connection handle.
uint16_t	report_length	Report Length.
uint8_t	hids_nb	HIDS Instance.
uint8_t	report_nb	Report Instance.

uint8_t	ind_type	Indication Type.
uint8_t	report	Report value.

Function Documentation

int app_hogprh_enable_cfm_handler (ke_msg_id_t const *msgid*, struct hogprh_enable_cfm * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HOGPRH_ENABLE_CFM
in	*param*	Pointer to the struct hogprh_enable_cfm
in	*dest_id*	TASK_APP
in	*src_id*	TASK_HOGPRH

Returns:

If the message was consumed or not.

Description:

This API message is used by the Report Host to either send the discovery results of HIDS on the HID device and confirm enabling of the Report Host role, or to simply confirm enabling of Report Host role if it is a normal connection and the attribute details are already known.

int app_hogprh_wr_char_rsp_handler (ke_msg_id_t const *msgid*, struct hogprh_char_req_rsp * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HOGPRH_WR_CHAR_RSP
in	*param*	Pointer to the struct hogprh_char_req_rsp
in	*dest_id*	TASK_APP
in	*src_id*	TASK_HOGPRH

Returns:

If the message was consumed or not.

Description:

This API is used to inform the application about the status of the writing request that has been sent.

int app_hogprh_proto_mode_rd_rsp_handler (ke_msg_id_t const *msgid*, struct hogprh_proto_mode_rd_rsp * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HOGPRH_PROTO_MODE_RD_RSP
in	*param*	Pointer to the struct hogprh_proto_mode_rd_rsp
in	*dest_id*	TASK_APP
in	*src_id*	TASK_HOGPRH

Returns:

If the message was consumed or not.

Description:

This API is used to inform the application about the read Protocol Mode Characteristic value.

int app_hogprh_report_map_rd_rsp_handler (ke_msg_id_t const *msgid*, struct hogprh_report_map_rd_rsp * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HOGPRH_REPORT_MAP_RD_RSP
in	*param*	Pointer to the struct hogprh_report_map_rd_rsp
in	*dest_id*	TASK_APP

in	*src_id*	TASK_HOGPRH

Returns:

If the message was consumed or not.

Description:

This API is used to inform the application about the read Report Map Characteristic value.

int app_hogprh_report_map_ext_rep_ref_rd_rsp_handler (ke_msg_id_t const *msgid*, struct hogprh_report_map_ext_rep_ref_rd_rsp * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HOGPRH_REPORT_MAP_EXT_REP_REF_RD_RSP
in	*param*	Pointer to the struct hogprh_report_map_ext_rep_ref_rd_rsp
in	*dest_id*	TASK_APP
in	*src_id*	TASK_HOGPRH

Returns:

If the message was consumed or not.

Description:

This API is used to inform the application about the read External Report Reference Descriptor value.

int app_hogprh_report_ref_rd_rsp_handler (ke_msg_id_t const *msgid*, struct hogprh_report_ref_rd_rsp * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HOGPRH_REPORT_REP_REF_RD_RSP
in	*param*	Pointer to the struct hogprh_report_ref_rd_rsp
in	*dest_id*	TASK_APP
in	*src_id*	TASK_HOGPRH

Returns:

If the message was consumed or not.

Description:

This API is used to inform the application about the read Report Reference Descriptor value.

int app_hogprh_hid_info_rd_rsp_handler (ke_msg_id_t const *msgid*, struct hogprh_hid_info_rd_rsp * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HOGPRH_HID_INFO_RD_RSP
in	*param*	Pointer to the struct hogprh_hid_info_rd_rsp
in	*dest_id*	TASK_APP
in	*src_id*	TASK_HOGPRH

Returns:

If the message was consumed or not.

Description:

This API is used to inform the application about the read HID Information Characteristic value.

int app_hogprh_cfg_ntf_rd_rsp_handler (ke_msg_id_t const *msgid*, struct hogprh_cfg_ntf_rd_rsp * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HOGPRH_NTF_CFG_RD_RSP
in	*param*	Pointer to the struct hogprh_cfg_ntf_rd_rsp
in	*dest_id*	TASK_APP
in	*src_id*	TASK_HOGPRH

Returns:

If the message was consumed or not.

Description:

The API is used to inform the application about the read Client Characteristic Configuration Descriptor value.

int app_hogprh_report_ind_handler (ke_msg_id_t const *msgid*, struct hogprh_report_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HOGPRH_REPORT_IND
in	*param*	Pointer to the struct hogprh_report_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_HOGPRH

Returns:

If the message was consumed or not.

Description:

This API is used to inform the application about the read Client Characteristic Configuration Descriptor value.

The following table presents all the possible values for the ind_type parameter:

- HOGPRH_IND_NTF (0x00): The Report Characteristic value has been received has a notification and the value is complete.
- HOGPRH_IND_RD_RSP (0x01): The Report Characteristic value has been received has a read response.
- HOGPRH_IND_INCOMPLETE_NTF (0x02): The Report Characteristic value has been received has a notification and the value is not complete. See the note below.

Note:

Here is an extract of the BLE HIDS specification, "Notification of characteristic values can contain at most [ATT_MTU-3] bytes of data by definition. Data beyond [ATT_MTU-3] bytes long is not included in a notification, and must instead be read using the GATT Read Long Characteristic Value sub-procedure. The possibility that data to be notified in a Report characteristic value could change before the HID Host completed an outstanding Read Long Characteristic Value sub-procedure, and therefore be lost, exists. For this reason it is strongly recommended that HID Devices support an ATT_MTU large enough to transfer their largest possible Report characteristic value in a single transaction."

Thus when an indication in received with an indication type set to HOGPRH_IND_INCOMPLETE_NTF, the application can begin to parse this incomplete Report value. Then it must wait for another indication whose the indication type will be set to HOGPRH_IND_RD_RSP and which will contain the whole Report Characteristic value (the first indication can be discarded if needed).

int app_hogprh_char_req_rsp_handler (ke_msg_id_t const *msgid*, struct hogprh_char_req_rsp * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HOGPRH_RD_CHAR_ERR_RSP
in	*param*	Pointer to the struct hogprh_char_req_rsp
in	*dest_id*	TASK_APP
in	*src_id*	TASK_HOGPRH

Returns:

If the message was consumed or not.

Description:

This handler is called when application sent a read request is not complient with the specification or the implementation limitations.

3.7 Heart Rate Profile

3.7.1 Heart Rate Profile Collector API

Detailed Description

HRPC role is meant to be activated on the device that will collect the Heart Rate measurements from the Heart Rate Sensor. It implies it is a GAP Central. The FW task for this role will discover the HRS present on the peer Server, after establishing connection, and will allow configuration of the HRS attributes if so required. This file contains the implementation of this API.

Function Documentation

void app_hrpc_enable_req (struct hrs_content * *hrs*, uint16_t *conhdl*)

Parameters:

in	*hrs*	Heart Rate Service Content Structure.
in	*conhdl*	Connection handle for which the profile Heart Rate Collector role is enabled.

Response:

HRPC_ENABLE_CFM

Description:

This API is used for enabling the Collector role of the Heart Rate profile. This function contains BLE connection handle, the connection type and the previously saved discovered HRS details on peer. The connection type may be 0 = Connection for discovery/initial configuration or 1 = Normal connection. This parameter is used by Application to discover peer device services once at first connection. Application shall save those information to reuse them for other connections. During normal connection, previously discovered device information can be reused.

This is useful since most use cases allow Heart Rate Sensor to disconnect the link once all measurements have been sent to Collector.

If it is a discovery /configuration type of connection, the HRS parameters are useless, they will be filled with 0's. Otherwise they will contain pertinent data which will be kept in the Collector environment while enabled. It allows for the Application to not be aware of attribute details.

For a normal connection, the response to this request is sent right away after saving the HRS content in the environment and registering HRPC in GATT to receive the indications and notifications for the known attribute handles in HRS that would be notified/indicated. For a discovery connection, discovery of the peer HRS is started and the response will be sent at the end of the discovery with the discovered attribute details.

void app_hrpc_rd_char_req (uint8_t *char_code*, uint16_t *conhdl*)

Parameters:

in	*char_code*	Code for which characteristic to read.
in	*conhdl*	Connection handle for which the profile Heart Rate Collector role is enabled.

Response:

HRPC_RD_CHAR_RSP or HRPC_ERROR_IND

Note:

char_code:

- HRPC_RD_HRS_HR_MEAS ///Read HRS Heart Rate Measurement
- HRPC_RD_HRS_BODY_SENSOR_LOC ///Body Sensor Location
- HRPC_RD_HRS_CNTL_POINT ///Heart Rate Control Point
- HRPC_RD_HRS_HR_MEAS_CFG ///Read HRS Heart Rate Measurement Client Cfg. Desc

Description:

This API is used by the application to send a GATT_READ_CHAR_REQ with the parameters deduced from the char_code. The definitions for the different mapping codes for characteristics that are possibly readable are in hrpc.h (for HRS). Upon reception of this message, HRPC checks whether the parameters are correct, then if the handle for the characteristic is valid (not 0x0000) and the request is sent to GATT. When the peer has responded to GATT, and the response is routed to HRPC, the HRPC_RD_CHAR_RSP message will be generically built and the Application must be able to interpret it based on the read request it made. And error status is also possible either for the Read procedure or for the application request, in the second case, the HRPC_ERROR_IND message is sent to Application.

void app_hrpc_cfg_indntf_req (uint16_t *cfg_val*, uint16_t *conhdl*)

Parameters:

in	*cfg_val*	Stop/notify/indicate value to configure into the peer characteristic.
in	*conhdl*	Connection handle for which the profile Heart Rate Collector role is enabled.

Response:

HRPC_WR_CHAR_RSP or HRPC_ERROR_IND

Note:

cfg_val:

- PRF_CLI_STOP_NTFIND
- PRF_CLI_START_NTF
- PRF_CLI_START_IND

Description:

This API is used by the application to send a GATT_WRITE_CHAR_REQ with the parameters deduced from the char_code and cfg_val. The definitions for the different codes for characteristics that can be configured to indicate/notify are in hrpc.h. Upon reception of this message, HRPC checks whether the parameters are correct, then if the handle for the characteristic is valid (not 0x0000) and the request is sent to GATT. When the peer has responded to GATT, and the response is routed to HRPC, the HRPC_WR_CHAR_RSP message will be generically built and sent to Application. An error status is also possible either for the Write procedure or for the application request, in the second case, the HRPC_ERROR_IND message is sent to Application.

void app_hrpc_wr_cntl_point_req (uint8_t *val*, uint16_t *conhdl*)

Parameters:

in	*val*	Reset(1).
in	*conhdl*	Connection handle for which the profile Heart Rate Collector role is enabled.

Response:

HRPC_WR_CHAR_RSP or HRPC_ERROR_IND

Description:

This API is used by the application to write control point attribute in order to reset Energy Expanded value.

3.7.2 Heart Rate Profile Collector Task API

Detailed Description

Heart Rate Profile Collector Task APIs are used to handle the message from HRPC or APP.

Data Structure Documentation

struct hrpc_enable_cfm

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	status	status
struct hrs_content	hrs	Existing handle values hrs.

struct hrpc_error_ind

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	status	Status.

struct hrpc_rd_char_rsp

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	status	Status.
struct att_info_data	data	Holder of retrieved data.

struct hrpc_wr_char_rsp

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	status	Status.

struct hrpc_meas_ind

Data Fields:

uint16_t	conhdl	Connection handle.
struct hrs_hr_meas	meas_val	Heart Rate measurement.

Function Documentation

int app_hrpc_enable_cfm_handler (ke_msg_id_t const *msgid*, struct hrpc_enable_cfm * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HRPC_ENABLE_CFM
in	*param*	Pointer to struct hrpc_enable_cfm
in	*dest_id*	TASK_APP
in	*src_id*	TASK_HRPC

Returns:

If the message was consumed or not.

Description:

This API is used by the Collector to either send the discovery results of HRS on the Heart Rate and confirm enabling of the Collector role, or to simply confirm enabling of Collector role if it is a normal connection and the attribute details are already known.

int app_hrpc_error_ind_handler (ke_msg_id_t const *msgid*, struct hrpc_error_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HRPC_ERROR_IND
in	*param*	Pointer to struct hrpc_error_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_HRPC

Returns:

If the message was consumed or not.

Description:

This API is used by the Collector role to inform the Application of an error occurred in different situations. The error codes are proprietary and defined in prf_types.h. An error may occur during attribute discovery or due to application request parameters. Following reception of this message, the application will decide the necessary action

int app_hrpc_rd_char_rsp_handler (ke_msg_id_t const *msgid*, struct hrpc_rd_char_rsp * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HRPC_RD_CHAR_RSP
in	*param*	Pointer to struct hrpc_rd_char_rsp
in	*dest_id*	TASK_APP
in	*src_id*	TASK_HRPC

Returns:

If the message was consumed or not.

Note:

Response for read Body Sensor Location and Heart Rate Measurement Client Cfg.Desc

Description:

This API message is used by the Collector role to inform the Application of a received read response. The status and the data from the read response are passed directly to Application, which must interpret them based on the request it made.

int app_hrpc_wr_char_rsp_handler (ke_msg_id_t const *msgid*, struct hrpc_wr_char_rsp * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HRPC_WR_CHAR_RSP
in	*param*	Pointer to struct hrpc_wr_char_rsp
in	*dest_id*	TASK_APP
in	*src_id*	TASK_HRPC

Returns:

If the message was consumed or not.

Description:

This API is used by the Collector role to inform the Application of a received write response. The status and the data from the write response are passed directly to Application, which must interpret them based on the request it made.

int app_hrpc_meas_ind_handler (ke_msg_id_t const *msgid*, struct hrpc_meas_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HRPC_HR_MEAS_IND
in	*param*	Pointer to struct hrpc_meas_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_HRPC

Returns:

If the message was consumed or not.

Note:

Heart Rate measurement structure refer to struct hrs_hr_meas

Description:

This API is used by the Collector role to inform the Application of a received Heart Rate value by notification. The application will do what it needs to do with the received measurement. No confirmation of reception is needed because the GATT sends it directly to the peer.

3.7.3 Heart Rate Profile Sensor

Detailed Description

The Bluetooth Low Energy Heart Rate profile enables the user to receive Heart Rate measurements from a Heart Rate sensor device and also configure it for different use cases. Within the profile, two roles can be supported: Collector and Sensor. The Heart Rate Sensor shall be a Server. The Collector shall be a Client.

Heart Rate Profile Sensor (HRPS): A HRPS (e.g. PC, phone, etc) is the term used by this profile to describe a device that can perform Heart Rate measurement and notify about on-going measurement and indicate final result to a peer BLE device.

Application needs manages multiple users configuration and storage of offline measurements.

Function Documentation

void app_hrps_create_db (uint8_t *features*)

Parameters:

in	*features*	Heart rate features used to create database, possible bit-mask values are: ● HRPS_BODY_SENSOR_LOC_CHAR_SUP ● HRPS_ENGY_EXP_FEAT_SUP

Response:

HRPS_CREATE_DB_CFM

Description:

This function shall be send after system power-on (or after GAP Reset) in order to create heart rate profile database. This database will be visible from a peer device but not usable until app_hrps_enable_req() is called within a BLE connection.

Note:

The Heart Rate profile requires the presence of one DIS characteristic

void app_hrps_enable_req (uint16_t *conhdl*, uint8_t *sec_lvl*, uint8_t *con_type*, uint16_t *hr_meas_ntf_en*, uint8_t *body_sensor_loc*)

Parameters:

in	*conhdl*	Connection handle for which the profile Heart Rate sensor role is enabled.
in	*sec_lvl*	Security level required for protection of HRS attributes: Service Hide and Disable are not permitted. Possible values are: ● PERM_RIGHT_ENABLE ● PERM_RIGHT_UNAUTH ● PERM_RIGHT_AUTH

in	*con_type*	Connection type: configuration(0) or discovery(1)
in	*hr_meas_ntf_en*	Heart Rate Notification configuration
in	*body_sensor_loc*	Body sensor leocation, Possible values are: ● HRS_LOC_OTHER ● HRS_LOC_CHEST ● HRS_LOC_WRIST ● HRS_LOC_FINGER ● HRS_LOC_HAND ● HRS_LOC_EAR_LOBE ● HRS_LOC_FOOT

Response:

None

Description:

This function is used for enabling the Heart Rate Sensor role of the Heart Rate profile. Before calling this function, a BLE connection shall exist with peer device. Application shall provide connection handle in order to activate the profile.

void app_hrps_measurement_send (uint16_t *conhdl*, struct hrs_hr_meas * *meas_val*)

Parameters:

in	*conhdl*	Connection handle for which the profile Heart Rate sensor role is enabled.
in	*meas_val*	Pointer to the struct hrs_hr_meas containing Heart Rate measurement value

Response:

HRPS_MEAS_SEND_CFM or None

Description:

This function is used by the application (which handles the Heart Rate device driver and measurements) to send a Heart Rate measurement through the Heart Rate sensor role

3.7.4 Heart Rate Profile Sensor Task API

Detailed Description

Heart Rate Profile Sensor Task APIs are used to handle the message from HRPS or APP.

Data Structure Documentation

struct hrps_create_db_cfm

Data Fields:

uint8_t	status	Status.

struct hrps_disable_ind

Data Fields:

uint16_t	conhdl	
uint16_t	hr_meas_ntf_en	Heart Rate Notification configuration.

struct hrps_cfg_indntf_ind

Data Fields:

uint16_t	conhdl	Connection handle.
uint16_t	cfg_val	Stop/notify/indicate value to configure into the peer characteristic.

struct hrps_meas_send_cfm

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	status	Status.

struct hrps_energy_exp_reset_ind

Data Fields:

uint16_t	conhdl	Connection handle.

Function Documentation

int app_hrps_create_db_cfm_handler (ke_msg_id_t const *msgid*, struct hrps_create_db_cfm * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HRPS_CREATE_DB_CFM
in	*param*	struct hrps_create_db_cfm
in	*dest_id*	TASK_APP
in	*src_id*	TASK_HRPS

Returns:

If the message was consumed or not.

Description:

This handler will be triggered after a database creation. It contains status of database creation.

int app_hrps_disable_ind_handler (ke_msg_id_t const *msgid*, struct hrps_disable_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HRPS_DISABLE_IND
in	*param*	Pointer to the struct hrps_disable_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_HRPS

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the Application of a correct disable. The configuration that the collector has set in HRS attributes must be conserved and the 4 values that are important are sent back to the application for safe keeping until the next time this profile role is enabled.

int app_hrps_error_ind_handler (ke_msg_id_t const *msgid*, struct prf_server_error_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HRPS_ERROR_IND
in	*param*	Pointer to the struct prf_server_error_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_HRPS

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the Application of an occurred error.

int app_hrps_send_means_cfm_handler (ke_msg_id_t const *msgid*, struct hrps_meas_send_cfm * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HRPS_MEAS_SEND_CFM
in	*param*	Pointer to the struct hrps_meas_send_cfm
in	*dest_id*	TASK_APP
in	*src_id*	TASK_HRPS

Returns:

If the message was consumed or not.

Description:

This handler is used to report to the application a confirmation, or error status of a notification request being sent by application.

int app_hrps_cfg_indntf_ind_handler (ke_msg_id_t const *msgid*, struct hrps_cfg_indntf_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HRPS_CFG_INDNTF_IND
in	*param*	Pointer to the struct hrps_cfg_indntf_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_HRPS

Returns:

If the message was consumed or not.

Description:

This handler is usedto inform application that peer device has changed notification configuration.

int app_hrps_energy_exp_reset_ind_handler (ke_msg_id_t const *msgid*, struct hrps_energy_exp_reset_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HRPS_ENERGY_EXP_RESET_IND
in	*param*	Pointer to the struct hrps_energy_exp_reset_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_HRPS

Returns:

If the message was consumed or not.

Description:

This handler is used to inform application that Energy Expanded value shall be reset.

3.8 Health Thermometer Profile

3.8.1 Health Thermometer Profile Collector API

Detailed Description

HTPC role is meant to be activated on the device that will collect the temperature measurements from the Thermometer. It implies it is a GAP Central. The FW task for this role will discover the HTS present on the peer Server, after establishing connection, and will allow configuration of the HTS attributes if so required. This file contains the implementation of this API.

Function Documentation

void app_htpc_enable_req (struct htpc_hts_content * *hts*, uint16_t *conhdl*)

Parameters:

in	*hts*	HTS details.
in	*conhdl*	Connection handle for which the profile Collector role is enabled.

Response:

HTPC_ENABLE_CFM

Description:

This API is used by Application to send message to TASK_HTPC for enabling the Collector role of the Health Thermometer profile, and it contains the connection handle for the connection this profile is activated, the connection type and the previously saved discovered HTS details on peer.

The connection type may be 0 = Connection for discovery/initial configuration or 1 = Normal connection. This difference has been made and Application would handle it in order to not discover the HTS on the Thermometer at every connection, but do it only once and keep the discovered details in the Collector device between connections. Configuration can be done during a normal connection also, but since most use cases allow Thermometer to disconnect the link once all measurements have been sent to Collector, the Collector may not have the time for it.

If it is a discovery /configuration type of connection, the hts and dis parameters are useless, they will be filled with 0's. Otherwise they will contain pertinent data which will be kept in the Collector environment while enabled. It allows for the Application to not be aware of attribute details.

For a normal connection, the response to this request is sent right away after saving the HTS and DIS content in the environment and registering HTPC in GATT to receive the indications and notifications for the known attribute handles in HTS that would be notified/indicated. For a discovery connection, discovery of the peer HTS is started and the response will be sent at the end of the discovery with the discovered attribute details.

void app_htpc_rd_char_req (uint8_t *char_code*, uint16_t *conhdl*)

Parameters:

in	*char_code*	Health Thermometer Service Characteristics ● HTPC_RD_HTS_TEMP_TYPE ● HTPC_RD_HTS_MEAS_INTV ● HTPC_RD_HTS_TEMP_MEAS_CLI_CFG ● HTPC_RD_HTS_INTM_TEMP_CLI_CFG ● HTPC_RD_HTS_MEAS_INTV_CLI_CFG ● HTPC_RD_HTS_MEAS_INTV_VAL_RGE
in	*conhdl*	Connection handle for which the profile Collector role is enabled.

Response:

HTPC_RD_CHAR_RSP or HTPC_ERROR_IND

Description:

This API is used by Application to send a GATT_READ_CHAR_REQ with the parameters deduced from the char_code. Upon reception of this message, HTPC checks whether the parameters are correct, then if the handle for the characteristic is valid (not 0x0000) and the request is sent to GATT. When the peer has responded to GATT, and the response is routed to HTPC, the HTPC_RD_CHAR_RSP message will be generically built and the Application must be able to interpret it based on the read request it made. And error status is also possible

either for the Read procedure or for the application request, in the second case, the HTPC_ERROR_IND message is sent to Application. No parsing intelligence of the received response is added in this API handler, so all the work of interpretation must be added in the Application depending of its request and use of the response.

void app_htpc_cfg_indntf_req (uint8_t *char_code*, uint16_t *cfg_val*, uint16_t *conhdl*)

Parameters:

in	*char_code*	Health Thermometer Service Characteristics
in	*cfg_val*	Possible values for setting client configuration characteristics
in	*conhdl*	Connection handle for which the profile Collector role is enabled.

Response:

HTPC_WR_CHAR_RSP or HTPC_ERROR_IND

Note:

char_code:

- HTPC_CHAR_HTS_TEMP_MEAS
- HTPC_CHAR_HTS_INTM_TEMP
- HTPC_CHAR_HTS_MEAS_INTV

cfg_val:

- PRF_CLI_STOP_NTFIND
- PRF_CLI_START_NTF
- PRF_CLI_START_IND

Description:

This API is used by the application to send a GATT_WRITE_CHAR_REQ with the parameters deduced from the char_code and cfg_val. The definitions for the different codes for characteristics that can be configured to indicate/notify are in htpc.h. Upon reception of this message, HTPC checks whether the parameters are correct, then if the handle for the characteristic is valid (not 0x0000) and the request is sent to GATT. When the peer has responded to GATT, and the response is routed to HTPC, the HTPC_WR_CHAR_RSP message will be generically built and sent to Application. An error status is also possible either for the Write procedure or for the application request, in the second case, the HTPC_ERROR_IND message is sent to Application.

void app_htpc_wr_meas_intv_req (uint16_t *intv*, uint16_t *conhdl*)

Parameters:

in	*intv*	range should between 1s ~ 65535s
in	*conhdl*	Connection handle for which the profile Collector role is enabled.

Response:

HTPC_WR_CHAR_RSP or HTPC_ERROR_IND

Description:

This API is used by the application to send a GATT_WRITE_CHAR_REQ to the HTS Measurement Interval Char. Upon reception of this message, HTPC checks whether the parameters are correct, then if the handle for the characteristic is valid (not 0x0000) and whether it is writable or not - if all OK, the request is sent to GATT, otherwise a HTPC_ERROR_IND message is built for the Application. When the peer has responded to GATT, and the response is routed to HTPC, the HTPC_WR_CHAR_RSP message will be generically built and sent to Application. An error status is also possible for the Write procedure, it will be sent through that same message. It is the application's responsibility to write a measurement interval value that respects the valid range in HTS characteristics.

3.8.2 Health Thermometer Profile Collector Task API

Detailed Description

Health Thermometer Profile Collector Task APIs are used to handle the message from HTPC or APP.

Data Structure Documentation

struct htpc_enable_cfm

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	status	status
struct htpc_hts_content	hts	HTS handle values and characteristic properties.

struct htpc_error_ind

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	status	Status.

struct htpc_rd_char_rsp

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	status	Status.
uint8_t	att_code	Att. Code.
struct att_info_data	data	Holder of retrieved data.

struct htpc_wr_char_rsp

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	status	Status.

struct htpc_temp_ind

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	flag_stable_meas	Stable or intermediary type of temperature.
struct htp_temp_meas	temp_meas	Temperature Measurement Structure.

struct htpc_meas_intv_ind

Data Fields:

uint16_t	conhdl	Connection handle.
uint16_t	intv	Interval.

Function Documentation

int app_htpc_enable_cfm_handler (ke_msg_id_t const *msgid*, struct htpc_enable_cfm * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HTPC_ENABLE_CFM
in	*param*	Pointer to struct htpc_enable_cfm
in	*dest_id*	TASK_APP
in	*src_id*	TASK_HTPC

Returns:

If the message was consumed or not.

Description:

This API is used by the Collector to either send the discovery results of HTS on the Thermometer and confirm enabling of the Collector role, or to simply confirm enabling of Collector role if it is a normal connection and the attribute details are already known.

int app_htpc_error_ind_handler (ke_msg_id_t const *msgid*, struct htpc_error_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HTPC_ERROR_IND
in	*param*	Pointer to struct htpc_error_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_HTPC

Returns:

If the message was consumed or not.

Description:

This API is used by the Collector role to inform the Application of an error occurred in different situations. The error codes are specific to this profile and defined in htpc.h. An error may occur during attribute discovery or due to application request parameters. Following reception of this message, the application will decide the necessary action.

int app_htpc_rd_char_rsp_handler (ke_msg_id_t const *msgid*, struct htpc_rd_char_rsp * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HTPC_RD_CHAR_RSP
in	*param*	Pointer to struct htpc_rd_char_rsp
in	*dest_id*	TASK_APP
in	*src_id*	TASK_HTPC

Returns:

If the message was consumed or not.

Description:

This API is used by the Collector role to inform the Application of a received read response. The status and the data from the read response are passed directly to Application, which must interpret them based on the request it made.

int app_htpc_wr_char_rsp_handler (ke_msg_id_t const *msgid*, struct htpc_wr_char_rsp * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HTPC_WR_CHAR_RSP
in	*param*	Pointer to struct htpc_wr_char_rsp
in	*dest_id*	TASK_APP
in	*src_id*	TASK_HTPC

Returns:

If the message was consumed or not.

Description:

This API is used by the Collector role to inform the Application of a received write response. The status and the data from the write response are passed directly to Application, which must interpret them based on the request it made.

int app_htpc_temp_ind_handler (ke_msg_id_t const *msgid*, struct htpc_temp_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HTPC_TEMP_IND
in	*param*	Pointer to struct htpc_temp_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_HTPC

Returns:

If the message was consumed or not.

Note:

get temperature value, 32bits float by conversion rule:

- first byte: exponent (signed)
- following three bytes: integer
- for example: 0xff000173 = 371*10(expo: -1) = 37.1 Celsius

Description:

This API is used by the Collector role to inform the Application of a received temperature value, either by notification (flag_stable_meas = intermediate) or indication (flag_stable_meas = stable). No confirmation of reception is needed because the GATT sends it directly to the peer.

int app_htpc_meas_intv_ind_handler (ke_msg_id_t const *msgid*, struct htpc_meas_intv_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HTPC_MEAS_INTV_IND
in	*param*	Pointer to struct htpc_meas_intv_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_HTPC

Returns:

If the message was consumed or not.

Description:

This API is used by the Collector role to inform the Application of a received Measurement Interval Char. Indication and the value it indicates. This value should be used by the Application as seen fit. No response is necessary (the GATT sends the necessary confirmation to the Indication PDU).

int app_htpc_disable_ind_handler (ke_msg_id_t const *msgid*, struct prf_client_disable_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HTPC_DISABLE_IND
in	*param*	Pointer to struct prf_client_disable_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_HTPC

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the application that the Health Thermometer Profile Client Role task has been correctly disabled or if an error has occurred during this process.

3.8.3 Health Thermometer Profile Thermometer

Detailed Description

An actual thermometer device does not exist on current platform, so measurement values that would come from a driver are replaced by simple counters sent at certain intervals following by the profile attributes configuration. When a measurement interval has been set to a non-zero value in a configuration connection, once reconnected, TH will send regular measurement INDs if Temp Meas Char Cfg is set to indicate and using the Meas Intv Value. The INDs will continue until meas interval is set to 0 or connection gets disconnected by C. Measurements should be stored even so, until profile is disabled.

If the measurement interval has been set to 0, then if Intermediate Temp is set to be notified and Temp Meas to be indicated, then a timer of fixed length simulates sending several NTF before and indication of a "stable" value. This fake behavior should be replaced once a real driver exists. If Intermediary Temp cannot be notified, just send the indication, if neither can be sent (the configuration connection should never leave this like this) then disconnect.

Function Documentation

void app_htpt_create_db (uint16_t *valid_range_min*, uint16_t *valid_range_max*, uint8_t *features*)

Parameters:

in	*valid_range_min*	Minimal measurement interval value
in	*valid_range_max*	Maximal measurement interval value
in	*features*	Indicate if optional features are supported or not. Value of this parameter shall be set using the following masks: • HTPT_TEMP_TYPE_CHAR_SUP • HTPT_INTERM_TEMP_CHAR_SUP • HTPT_MEAS_INTV_CHAR_SUP • HTPT_MEAS_INTV_IND_SUP • HTPT_MEAS_INTV_WR_SUP

Response:

HTPT_CREATE_DB_CFM

Description:

This function shall be used to add an instance of the Health Thermometer service into the database. This should be done during the initialization phase of the device.

void app_htpt_enable_req (uint16_t *conhdl*, uint8_t *sec_lvl*, uint8_t *con_type*, uint16_t *temp_meas_ind_en*, uint16_t *interm_temp_ntf_en*, uint16_t *meas_intv_ind_en*, uint16_t *meas_intv*)

Parameters:

in	*conhdl*	Connection handle for which the profile Thermometer role is enabled
in	*sec_lvl*	Security level required for protection of HTS attributes • PERM_RIGHT_ENABLE • PERM_RIGHT_UNAUTH • PERM_RIGHT_AUTH
in	*con_type*	Connection type: configuration(PRF_CON_DISCOVERY) or normal(PRF_CON_NORMAL)
in	*temp_meas_ind_en*	Value stored for Temperature Measurement Client Configuration Characteristic
in	*interm_temp_ntf_en*	Value stored for Intermediate Temperature Client Configuration Characteristic

in	*meas_intv_ind_en*	Value stored for Measurement Interval Client Configuration Characteristic
in	*meas_intv*	Stored Measurement Interval value

Response:

None

Description:

This function is used for enabling the Thermometer role of the Health Thermometer profile.

void app_htpt_temp_send (uint16_t *conhdl*, struct htp_temp_meas * *temp_meas*, uint8_t *flag_stable_meas*)

Parameters:

in	*conhdl*	Connection handle for which the profile Thermometer role is enabled
in	*temp_meas*	Pointer to the struct htp_temp_meas containing Temperature Measurement value
in	*flag_stable_meas*	Indicate if the temperature measurement is stable: 0 (will be sent using the Temperature Measurement characteristic) or not: 1 (will be sent using the Intermediate Temperature characteristic)

Response:

HTPT_TEMP_SEND_CFM or none

Description:

This function is used by the application (which handles the temperature device driver and measurements) to send a temperature measurement through the Thermometer role.

Note:

Message HTPT_CFG_INDNTF_IND will be received as a hint to call this function.

void app_htpt_measurement_intv_send (uint16_t *conhdl*, uint16_t *meas_intv*)

Parameters:

in	*conhdl*	Connection handle for which the profile Thermometer role is enabled.
in	*meas_intv*	Measurement Interval value.

Response:

None

Description:

This function is used by the application to order the HTPT profile to generate an indication (if enabled) of the Measurement Interval Char. This can be done as the application desires, at each connection, or if the measurement interval value has been modified locally (interface for this is not provided since a normal thermometer would have very few configurable UI elements and configuration should be done through Collector).

void app_htpt_temp_type_send (uint8_t *value*)

Parameters:

in	*value*	new temperature type value

Response:

None

Description:

This function is used by the application to update the value of the temperature type characteristic.

3.8.4 Health Thermometer Profile Thermometer Task API

Detailed Description

Health Thermometer Profile Thermometer Task APIs are used to handle the message from HTPT or APP.

Data Structure Documentation

struct htpt_create_db_cfm

Data Fields:

uint8_t	status	Status.

struct htpt_disable_ind

Data Fields:

uint16_t	conhdl	Connection handle.
uint16_t	temp_meas_ind_en	Temperature measurement indication configuration.
uint16_t	interm_temp_ntf_en	Intermediate temperature notification configuration.
uint16_t	meas_intv_ind_en	Measurement interval indication configuration.
uint16_t	meas_intv	Measurement interval.

struct htpt_temp_send_cfm

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	status	Status.
uint8_t	cfm_type	Confirmation Type.

struct htpt_meas_intv_chg_ind

Data Fields:

uint16_t	intv	

struct htpt_cfg_indntf_ind

Data Fields:

uint16_t	conhdl	Connection handle.
uint16_t	cfg_val	Stop/notify/indicate value to configure into the peer characteristic.
uint8_t	char_code	Own code for differentiating between Temperature Measurement, Intermediate Temperature and Measurement Interval chars.

Function Documentation

int app_htpt_create_db_cfm_handler (ke_msg_id_t const *msgid*, struct htpt_create_db_cfm * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HTPT_CREATE_DB_CFM
in	*param*	Pointer to the struct htpt_create_db_cfm
in	*dest_id*	TASK_APP
in	*src_id*	TASK_HTPT

Returns:

If the message was consumed or not.

Description:

This handler will be called after reception of a database creation. The status parameter indicates if the HTS has been successfully added (ATT_ERR_NO_ERROR) or not (ATT_INSUFF_RESOURCE).

int app_htpt_disable_ind_handler (ke_msg_id_t const *msgid*, struct htpt_disable_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HTPT_DISABLE_IND
in	*param*	Pointer to the struct htpt_disable_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_HTPT

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the Application of a correct disable. The configuration that the collector has set in HTS attributes must be conserved and the 4 values that are important are sent back to the application for safe keeping until the next time this profile role is enabled.

int app_htpt_error_ind_handler (ke_msg_id_t const *msgid*, struct prf_server_error_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HTPT_ERROR_IND
in	*param*	Pointer to the struct prf_server_error_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_HTPT

Returns:

If the message was consumed or not.

Description:

This handler is to inform the Application of an occurred error.

int app_htpt_temp_send_cfm_handler (ke_msg_id_t const *msgid*, struct htpt_temp_send_cfm * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HTPT_TEMP_SEND_CFM
in	*param*	Pointer to the struct htpt_temp_send_cfm
in	*dest_id*	TASK_APP
in	*src_id*	TASK_HTPT

Returns:

If the message was consumed or not.

Description:

This handler will be called when the status of a notification request sent by application for the Intermediate Temperature Char return. The importance of a confirmation of a specific indication is that the application can erase the stable temperature values that it sent the peer successfully. For intermediate temperatures the confirmation is useless, so they would not be stored.

int app_htpt_meas_intv_chg_ind_handler (ke_msg_id_t const *msgid*, struct htpt_meas_intv_chg_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HTPT_MEAS_INTV_CHG_IND
in	*param*	Pointer to the struct htpt_meas_intv_chg_ind
in	*dest_id*	TASK_APP

in	*src_id*	TASK_HTPT

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the application that the measurement interval value has changed. The application uses the new value to either decide to stop periodic measurements if the value of the interval has changed from non 0 to 0, or the opposite, to start periodic measurements using the interval value, if the value has changed from 0 to non 0. This handler will only be triggered if the new value that the Collector is trying to write is valid (within the Valid Range descriptor minimum and maximum values). If the value is not within range, this handler is never be triggered by the application because the HTPT will send an Error Response to the Collector with the 'Out of Range' code 0x80 and the new value will never be set.

int app_htpt_cfg_indntf_ind_handler (ke_msg_id_t const *msgid*, struct htpt_cfg_indntf_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	HTPT_CFG_INDNTF_IND
in	*param*	Pointer to the struct htpt_cfg_indntf_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_HTPT

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the Application of a new value set in one of the 3 Client Characteristic Configuration Descriptors in HTS. It allows the application to be aware of its current settings for HTS and to alter its behavior accordingly if the implementation desires it.

3.9 Proximity Profile

3.9.1 Proximity Monitor API

Detailed Description

PEOXM role is meant to be activated on the device that will monitor the connection with the Reporter device. It implies it is a GAP Central. The FW task for this role will discover the LLS, IAS and TPS services present on the peer Server, after establishing connection, and will allow writing different alert levels to the Alert Level characteristic in the LLS or IAS, and also reading the Tx Power Level characteristic in TPS.

Function Documentation

void app_proxm_enable_req (struct svc_content * *lls*, struct svc_content * *ias*, struct svc_content * *txps*, uint16_t *conhdl*)

Parameters:

in	*lls*	Link Loss Service information.
in	*ias*	Immediate Alert Service information.
in	*txps*	Tx Power Service information.
in	*conhdl*	Connection handle for which the profile Monitor role is enabled.

Response:

PROXM_ENABLE_CFM or PROXM_ERROR_IND

Description:

The API is used for enabling the Monitor role of the Proximity profile. This function contains the connection handle for the connection this profile is activated, the connection type and the previously saved discovered LLS, IAS and TPS details on peer.

The connection type may be 0 = Connection for discovery or 1 = Normal connection. This difference has been made and Application would handle it in order to not discover the attributes on the Reporter at every connection, but do it only once and keep the discovered details in the Monitor device between connections.

If it is a discovery type connection, the LLS, IAS and TPS parameters are useless, they will be filled with 0's. Otherwise it will contain pertinent data which will be kept in the Monitor environment while enabled. It allows for the Application to not be aware of attribute details, and only give them to the profile role from a storage area where they are kept in between connections (NVDS...).

For a normal connection, the response to this request is sent right away after saving the lls, ias and tps content in the environment. For a discovery connection, discovery of the peer attributes is started and the response will be sent at the end of the discovery with the discovered attribute details. The discovery starts with the 3 services, then with the characteristics of the discovered services. If an error happens during discovery (mandatory conditions present in the specification are not respected - missing services, mandatory characteristics...) it is signaled right away to the application setting an appropriate value in the status parameter of the response.

void app_proxm_rd_alert_lvl_req (uint16_t *conhdl*)

Parameters:

in	*conhdl*	Connection handle for which the profile Monitor role is enabled.

Response:

PROXM_RD_CHAR_RSP or PROXM_ERROR_IND

Description:

This API is used for reading the alert level in LLS Alert Level Characteristic. Upon reception, the Monitor checks if a valid handle for this characteristic is known in the monitor environment, otherwise sending a PROXM_ERROR_IND with error code 0x02. If the characteristic handle in the environment is valid, a read characteristic request is built for TASK_GATT using the LLS service and characteristic handle values. When the read response is received from GATT, it is sent to the Application to be analyzed.

void app_proxm_wr_alert_lvl_req (uint8_t *svc_code*, uint8_t *lvl*, uint16_t *conhdl*)

Parameters:

in	*svc_code*	code for the service in which the alert level should be written.
in	*lvl*	Alert level.
in	*conhdl*	Connection handle for which the profile Monitor role is enabled.

Response:

PROXM_WR_CHAR_RSP or PROXM_ERROR_IND

Note:

svc_code:

- PROXM_SET_LK_LOSS_ALERT ///Code for LLS Alert Level Char.
- PROXM_SET_IMMDT_ALERT ///Code for IAS Alert Level Char.

lvl:

- PROXM_ALERT_NONE

- PROXM_ALERT_MILD
- PROXM_ALERT_HIGH

Description:

This API is used by the application to set either a LLS Alert Level or an IAS Alert Level. Since these two service have characteristics of the same type (but not the same properties - LLS one is R&W, IAS one is Write no Response only), one API message for a very similar purpose was considered sufficient, and therefore a simple byte for differentiating whether the alert code should be set in LLS or IAS characteristic is used.

Upon reception of this request, the connection handle is checked, then the alert level to set in order to ensure a valid value, and then the svc_code for 0 or 1.. In case any of these checks fail, a PROXM_ERROR_IND message is sent to the Application with error code 0x01.

If the checks are successful, a write characteristic request is build for TASK_GATT, with the appropriate type (Simple Write or Write no response) and using the characteristic handle for the service indicated by svc_code. When the Write Response is received from TASK_GATT, it is sent to the Application for it to check the status.

void app_proxm_rd_txpw_lvl_req (uint16_t *conhdl*)

Parameters:

in	*conhdl*	Connection handle for which the profile Monitor role is enabled.

Response:

PROXM_RD_CHAR_RSP or PROXM_ERROR_IND

Description:

This API is used for reading the tx power level in TPS Tx Power Level Characteristic. Upon reception, the Monitor checks if a valid handle for this characteristic is known in the monitor environment, otherwise sending a PROXM_ERROR_IND with error code 0x02. If the characteristic handle in the environment is valid, a read characteristic request is built for TASK_GATT using the TPS service and characteristic handle values. When the read response is received from GATT, it is sent to the Application to be analyzed.

3.9.2 Proximity Monitor Task API

Detailed Description

Proximity Monitor Task APIs are used to handle the message from PROXM or APP.

Data Structure Documentation

struct proxm_enable_cfm

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	status	Status.
struct svc_content	lls	Reporter LLS details to keep in APP.
struct svc_content	ias	Reporter IAS details to keep in APP.
struct svc_content	txps	Reporter TPS details to keep in APP.

struct proxm_rd_char_rsp

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	status	Read characteristic response status code, may be GATT code or ATT error code.
uint8_t	char_code	Char. Code.

uint8_t	val	Value.

struct proxm_wr_char_rsp

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	status	Write characteristic response status code, may be GATT code or ATT error code.

Function Documentation

int app_proxm_enable_cfm_handler (ke_msg_id_t const *msgid*, struct proxm_enable_cfm * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	PROXM_ENABLE_CFM
in	*param*	Pointer to struct proxm_enable_cfm
in	*dest_id*	TASK_APP
in	*src_id*	TASK_PROXM

Returns:

If the message was consumed or not.

Note:

LLS is Mandatory IAS and TPS are optional

Description:

This API is used by the Monitor to either send the discovery results of LLS, IAS and TPS on Reporter and confirm enabling of the Monitor role, or to simply confirm enabling of the Monitor role if it is a normal connection and the LLS, IAS and TPS details are already known.

int app_proxm_rd_char_rsp_handler (ke_msg_id_t const *msgid*, struct proxm_rd_char_rsp * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	PROXM_RD_CHAR_RSP
in	*param*	Pointer to struct proxm_rd_char_rsp
in	*dest_id*	TASK_APP
in	*src_id*	TASK_PROXM

Returns:

If the message was consumed or not.

Description:

This API is used by the Monitor role to send the Application the characteristic read response data and the status of the read characteristic request. The application is in charge of deciphering the data or of the next step if an error is received.

int app_proxm_wr_char_rsp_handler (ke_msg_id_t const *msgid*, struct proxm_wr_char_rsp * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	PROXM_WR_CHAR_RSP
in	*param*	Pointer to struct proxm_wr_char_rsp
in	*dest_id*	TASK_APP
in	*src_id*	TASK_PROXM

Returns:

If the message was consumed or not.

Description:

This API is used by the Monitor role to send the Application the status of a characteristic write request, received in a response from TASK_GATT. The application will decide what to do if an error is received.

int app_proxm_disable_ind_handler (ke_msg_id_t const *msgid*, struct prf_client_disable_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	PROXM_DISABLE_IND
in	*param*	Pointer to struct prf_client_disable_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_PROXM

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the application that the Monitor role of the Proximity profile has been correctly disabled or if an error has occurred during this process.

3.9.3 Proximity Reporter

Detailed Description

The Proximity profile defines the behavior when a device moves away from a peer device so that the connection is dropped or the path loss increases above a preset level, causing an immediate alert. This alert can be used to notify the user that the devices have become separated. As a consequence of this alert, a device may take further action, for example to lock one of the devices so that it is no longer usable. The Proximity profile can also be used to define the behavior when the two devices come closer together such that a connection is made or the path loss decreases below a preset level. Within the profile, two roles can be supported: Monitor and Reporter. The Proximity Reporter shall be a server. The Proximity Monitor shall be a GATT client.

The Proximity Reporter device must have an instance of the Link Loss Service(LLS), and may also have the Immediate Alert Service(IAS) and Tx Power Service(TPS) in its attribute database. The two last ones must be used together, if one is missing, the other one should be ignored.

The LLS allows the user to set an alert level in the Reporter, which will be used by the reporter to alert in the corresponding way if the link is lost. The disconnection must not come voluntarily from one of the two devices in order to trigger the alert.

The IAS allows the user to set an immediate alert level based on path loss computation using the read Tx Power Level and RSSI monitored on received packets. According to the alert level set in IAS, the Reporter will start alerting immediately.

The TPS allows the user to read the Tx Power Level for the physical layer. The value is used by the Monitor to continuously evaluate path loss during the connection, and decide to trigger/stop an alert based on path loss going over/under a set threshold in the Monitor application.

Function Documentation

void app_proxr_create_db (uint8_t *features*)

Parameters:

in	*features*	Indicate if optional features is supported or not, possible values are: PROXR_IAS_TXPS_NOT_SUP PROXR_IAS_TXPS_SUP

Response:

PROXR_CREATE_DB_CFM

Description:

This function shall be used to after reception of create database. This should be done during the initialization phase of the device. The status parameter indicates if the services have been successfully added (ATT_ERR_NO_ERROR) or not (ATT_INSUFF_RESOURCE).

void app_proxr_enable_req (uint16_t *conhdl*, uint8_t *sec_lvl*, uint8_t *lls_alert_lvl*, int8_t *txp_lvl*)

Parameters:

in	*conhdl*	Connection handle for which the profile Reporter role is enabled
in	*sec_lvl*	Security level required for protection of attributes. Service Hide and Disable are not permitted. Possible values are: ● PERM_RIGHT_ENABLE ● PERM_RIGHT_UNAUTH ● PERM_RIGHT_AUTH
in	*lls_alert_lvl*	Saved value for LLS alert level, from previous profile use. 0 if the connection is a configuration connection.
in	*txp_lvl*	TX Power level, range from -100 to 20

Response:

None

Description:

This function is used for enabling the Reporter role of the Proximity profile. After calling this function, the services are unhidden from the peer discove.

3.9.4 Proximity profile Reporter Task API

Detailed Description

Proximity profile Reporter Task APIs are used to handle the message from PROXR or APP.

Data Structure Documentation

struct proxr_create_db_cfm

Data Fields:

uint8_t	status	Status.

struct proxr_disable_ind

Data Fields:

uint16_t	conhdl	Connection Handle.
uint8_t	lls_alert_lvl	LLS alert level to save in APP.

struct proxr_alert_ind

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	alert_lvl	Alert level.
uint8_t	char_code	Char Code - Indicate if IAS or LLS.

Function Documentation

int app_proxr_create_db_cfm_handler (ke_msg_id_t const *msgid*, struct proxr_create_db_cfm * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	PROXR_CREATE_DB_CFM
in	*param*	Pointer to the struct proxr_create_db_cfm
in	*dest_id*	TASK_APP
in	*src_id*	TASK_PROXR

Returns:

If the message was consumed or not.

Description:

This handler will be called after reception of create database. The status parameter indicates if the services have been successfully added (ATT_ERR_NO_ERROR) or not (ATT_INSUFF_RESOURCE).

int app_proxr_disable_ind_handler (ke_msg_id_t const *msgid*, struct proxr_disable_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	PROXR_DISABLE_IND
in	*param*	Pointer to the struct proxr_disable_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_PROXR

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the Application of the correct disable or the Reporter role, and to give the application the LLS alert level to save until the next activation of the Reporter role.

int app_proxr_alert_ind_handler (ke_msg_id_t const *msgid*, struct proxr_alert_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	PROXR_ALERT_IND
in	*param*	Pointer to the struct proxr_alert_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_PROXR

Returns:

If the message was consumed or not.

Description:

This handler is used to request the Application to start the alert on the device considering the indicated alert level. The handler may be triggered on two conditions: The IAS alert level characteristic has been written to a valid value, in which case alert_lvl will be set to the IAS alert level value. A disconnection with a reason other than the normal local/remote link terminations has been received, in which case alert_lvl will be set to the LLS alert level value. The Application actions following reception of this indication are strictly implementation specific (it may try to reconnect to the peer and stop alert upon that, or timeout the alert after acertain time, please see the specification)

3.10 Scan Parameter Profile

3.10.1 Scan Parameters Profile Client API

Detailed Description

Scan Parameters Profile Client APIs are used by APP to enable/disable Scan Parameters Profile Client Role, to read/write Scan Refresh Notification Configuration Value.

Function Documentation

void app_scppc_enable_req (uint16_t *scan_intv*, uint16_t *scan_window*, struct scps_content * *scps*, uint16_t *conhdl*)

Parameters:

in	*scan_intv*	Last Scan Interval value to write after discovery.
in	*scan_window*	Last Scan Window value to write after discovery.
in	*scps*	Scan Parameters Service Content Structure.
in	*conhdl*	Connection handle for which the profile client role is enabled.

Response:

SCPPC_ENABLE_CFM

Description:

This API is used for enabling the Client role of the SCPP. This Function contains BLE connection handle, the connection type and the previously saved discovered SCPS details on peer. The connection type may be PRF_CON_DISCOVERY (0x00) for discovery/initial configuration or PRF_CON_NORMAL (0x01) for a normal connection with a bonded device. Application shall save those information to reuse them for other connections. During normal connection, previously discovered device information can be reused. If it is a discovery/configuration type of connection, it is useless to fill the scps parameter are useless. Otherwise they will contain pertinent data which will be kept in the Client environment while enabled.

For a normal connection, the response to this request is sent right away after saving the SCPS content in the environment and registering SCPPC in GATT to receive the notifications for the known attribute handles in SCPS that would be notified (Scan Refresh Characteristic).

For a discovery connection, discovery of the peer SCPS is started and the response will be sent at the end of the discovery with the discovered attribute details.

void app_scppc_scan_intv_wd_wr_req (uint16_t *scan_intv*, uint16_t *scan_window*, uint16_t *conhdl*)

Parameters:

in	*scan_intv*	Scan Interval value.
in	*scan_window*	Scan Window value.
in	*conhdl*	Connection handle for which the profile client role is enabled.

Response:

SCPPC_WR_CHAR_RSP or SCPPC_ERROR_IND

Description:

This API shall be used to inform the Scan Server that the Scan Client has changed its intended scanning behavior. It will write the Scan Interval Window Characteristic value in the Scan Server database.

The provided scan parameters are saved within the role task environement so that when the Scan Server requires the latest scan parameters (sends a notification for the Scan Refresh Characteristic), these values are automatically written in its database.

void app_scppc_scan_refresh_ntf_cfg_rd_req (uint16_t *conhdl*)

Parameters:

in	*conhdl*	Connection handle for which the profile client role is enabled.

Response:

SCPPC_SCAN_REFRESH_NTF_CFG_RD_RSP or SCPPC_ERROR_IND

Description:

This API shall be used to read the value of the Scan Refresh Characteristic Client Characteristic Configuration Descriptor.

void app_scppc_wr_meas_intv_req (uint16_t *ntf_cfg*, uint16_t *conhdl*)

Parameters:

in	*ntf_cfg*	Notification Configuration.
in	*conhdl*	Connection handle for which the profile client role is enabled.

Response:

SCPPC_WR_CHAR_RSP or SCPPC_ERROR_IND

Note:

ntf_cfg:

- PRF_CLI_STOP_NTFIND
- PRF_CLI_START_NTF
- PRF_CLI_START_IND

Description:

This API shall be used to either enable or disable notifications for the Scan Refresh Characteristic.

int app_scppc_disable_ind_handler (ke_msg_id_t const *msgid*, struct prf_client_disable_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	SCPPC_DISABLE_IND
in	*param*	Pointer to struct prf_client_disable_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_SCPPC

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the application that the Client role of the SCPS has been correctly disabled or if an error has occurred during this process.

3.10.2 Scan Parameters Profile Client Task API

Detailed Description

Scan Parameters Profile Client Task APIs are used to handle the message from SCPPC or APP.

Data Structure Documentation

struct scppc_enable_cfm

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	status	status
struct scps_content	scps	Existing handle values scps.

struct scppc_error_ind

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	status	Status.

struct scppc_wr_char_rsp

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	status	Status.

struct scppc_scan_refresh_ntf_cfg_rd_rsp

Data Fields:

uint16_t	conhdl	Connection handle.
uint16_t	ntf_cfg	Notification COnfiguration Value.

Function Documentation

int app_scppc_enable_cfm_handler (ke_msg_id_t const *msgid*, struct scppc_enable_cfm * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	SCPPC_ENABLE_CFM
in	*param*	Pointer to struct scppc_enable_cfm
in	*dest_id*	TASK_APP
in	*src_id*	TASK_SCPPC

Returns:

If the message was consumed or not.

Description:

This API is used by the Client role task to either send the discovery results of SCPS on the peer device and confirm enabling of the Client role, or to simply confirm enabling of Client role if it is a normal connection and the attribute details are already known.

int app_scppc_error_ind_handler (ke_msg_id_t const *msgid*, struct scppc_error_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	SCPPC_ERROR_IND
in	*param*	Pointer to struct scppc_error_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_SCPPC

Returns:

If the message was consumed or not.

Description:

This API is called by the application when an error has been raised in the SCPP Client role task.

int app_scppc_wr_char_rsp_handler (ke_msg_id_t const *msgid*, struct scppc_wr_char_rsp * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	SCPPC_WR_CHAR_RSP
in	*param*	Pointer to struct scppc_wr_char_rsp
in	*dest_id*	TASK_APP
in	*src_id*	TASK_SCPPC

Returns:

If the message was consumed or not.

Description:

This API is called by the application when a write response has been received from the peer device after sending of a write request

int app_scppc_scan_refresh_ntf_cfg_rd_rsp_handler (ke_msg_id_t const *msgid*, struct scppc_scan_refresh_ntf_cfg_rd_rsp * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	SCPPC_SCAN_REFRESH_NTF_CFG_RD_RSP
in	*param*	Pointer to struct scppc_scan_refresh_ntf_cfg_rd_rsp
in	*dest_id*	TASK_APP
in	**src_id**	**TASK_SCPPC**

Returns:

If the message was consumed or not.

Description:

This API is called by the application to inform it about the read Client Characteristic Configuration Descriptor value for the Scan Refresh Characteristic.

3.10.3 Scan Parameters Profile Server

Detailed Description

The Bluetooth Low Energy Scan Parameters Profile is used to provide devices with information to assist them in managing their connection idle timeout and advertising parameters to optimize for power consumption and/or reconnection latency. Within the profile, two roles can be supported: Client and Server. The Scan Server shall be a server. The Scan Client shall be a client.

Function Documentation

void app_scpps_create_db (uint8_t *features*)

Parameters:

in	*features*	Indicate if scan fresh function is supported or not, possible values are: ● SCPPS_SCAN_REFRESH_CHAR_NOT_SUP ● SCPPS_SCAN_REFRESH_CHAR_SUP

Response:

SCPPS_CREATE_DB_CFM

Description:

This function is used to add one instance of the Scan Parameters Service in the database.

void app_scpps_enable_req (uint16_t *conhdl*, uint8_t *sec_lvl*, uint8_t *con_type*, uint16_t *scan_refresh_ntf_en*)

Parameters:

in	*conhdl*	Connection handle for which the profile scan parameters server role is enabled.
in	*sec_lvl*	Security level required for protection of SCPS attributes: Service Hide and Disable are not permitted. Possible values are: ● PERM_RIGHT_ENABLE ● PERM_RIGHT_UNAUTH ● PERM_RIGHT_AUTH

in	*con_type*	Connection type: configuration(0) or discovery(1)
in	*scan_refresh_ntf_en*	Scan Refresh Notification Configurations, possible values are: ● PRF_CLI_STOP_NTFIND ● PRF_CLI_START_NTF

Response:

None or SCPPS_EORROR_IND

Description:

This function shall be used after the connection with a peer device has been established in order to enable the SCPP Server role task for the specified connection.

void app_scpps_scan_refresh_req (uint16_t *conhdl*, uint8_t *scan_refresh*)

Parameters:

in	*conhdl*	Connection handle for which the profile scan parameters server role is enabled.
in	*scan_refresh*	Scan Refresh Value, 0 means Server requires refresh, 1 ~ 255 reserved for future use

Response:

SCPPS_SCAN_REFRESH_SEND_CFM

Description:

This function is used notify the Client that the Server writes the latest intended scan parameters to the Scan Interval Window Characteristic.

3.10.4 Scan Paramters Profile Server Task API

Detailed Description

Scan Paramters Profile Server Task APIs are used to handle the message from SCPPS or APP.

Data Structure Documentation

struct scpps_create_db_cfm

Data Fields:

uint8_t	status	Status about DB creation.

struct scpps_scan_intv_wd_ind

Data Fields:

uint16_t	conhdl	Connection handle.
struct scan_intv_wd	scan_intv_wd	Scan Interval Window.

struct scpps_scan_refresh_ntf_cfg_ind

Data Fields:

uint16_t	conhdl	Connection handle.
uint16_t	scan_refresh_ntf_en	Scan Refresh Notification Configuration.

struct scpps_scan_refresh_send_cfm

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	status	Status.

Function Documentation

int app_scpps_create_db_cfm_handler (ke_msg_id_t const *msgid*, struct scpps_create_db_cfm * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	SCPPS_CREATE_DB_CFM
in	*param*	Pointer to the struct scpps_create_db_cfm
in	*dest_id*	TASK_APP
in	*src_id*	TASK_SCPPS

Returns:

If the message was consumed or not.

Description:

This handler will be called after a database creation. It contains status of database creation.

int app_scpps_disable_ind_handler (ke_msg_id_t const *msgid*, struct scpps_disable_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	SCPPS_DISABLE_IND
in	*param*	Pointer to the struct scpps_disable_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_SCPPS

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the application of a corrent disable. The current notification confiuration value for the Scan Refresh characteristic are included in the paramerers so that the higher application may safely keep the configuration until the next time the profile is enabled.

int app_scpps_error_ind_handler (ke_msg_id_t const *msgid*, struct prf_server_error_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	SCPPS_ERROR_IND
in	*param*	Pointer to the struct prf_server_error_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_SCPPS

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the Application of occurred error information

int app_scpps_scan_intv_wd_ind_handler (ke_msg_id_t const *msgid*, struct scpps_scan_intv_wd_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	SCPPS_SCAN_INTV_WD_IND
in	*param*	Pointer to the struct scpps_scan_intv_wd_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_SCPPS

Returns:

If the message was consumed or not.

Description:

This handler will be triggered when the Scan Interval Window Characteristic value has been written by the peer device.

int app_scpps_scan_refresh_ntf_cfg_ind_handler (ke_msg_id_t const *msgid*, struct scpps_scan_refresh_ntf_cfg_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	SCPPS_SCAN_REFRESH_NTF_CFG_IND
in	*param*	Pointer to the struct scpps_scan_refresh_ntf_cfg_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_SCPPS

Returns:

If the message was consumed or not.

Description:

This handler will be triggered when the peer device has enabled or disabled sending of notiications for the Scan Refresh Characteristic.

int app_scpps_send_scan_refresh_cfm_handler (ke_msg_id_t const *msgid*, struct scpps_scan_refresh_send_cfm * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	SCPPS_SCAN_REFRESH_SEND_CFM
in	*param*	Pointer to the struct scpps_scan_refresh_send_cfm
in	*dest_id*	TASK_APP
in	*src_id*	TASK_SCPPS

Returns:

If the message was consumed or not.

Description:

This handler will be triggered if the Scan Refresh Characteristic value has been notified or not.

3.11 Time Profile

3.11.1 Time Profile Client API

Detailed Description

TIPC role is meant to be activated on the device that will collect the time values and information from the Time Server. It implies it is a GAP Central. The FW task for this role will discover the CTS (Mandatory), the NDCS (Optional) and the RTUS (Optional) present on the peer Server, after establishing connection, and will allow configuration of the CTS and RTUS attributes if so required. This file contains the implementation of this API.

Function Documentation

void app_tipc_enable_req (struct tipc_cts_content * *cts*, struct tipc_ndcs_content * *ndcs*, struct tipc_rtus_content * *rtus*, uint16_t *conhdl*)

Parameters:

in	*cts*	Current Time Service Structure.
in	*ndcs*	Next DST Change Service Structure.
in	*rtus*	Reference Time Update Service Structure.
in	*conhdl*	Connection handle for which the Time Client role is enabled.

Response:

TIPC_ENABLE_CFM or TIPC_ERROR_IND

Description:

This API is used for enabling the Client role of the Time profile. This Function contains BLE connection handle, the connection type and the previously saved discovered CTS, NDCS and RTUS details on peer.

The connection type may be PRF_CON_DISCOVERY for discovery/initial connection or PRF_CON_NORMAL for normal connection.

For a discovery connection, discovery of the peer CTS, NDCS and RTUS is started and the response will be sent at the end of the discovery with the discovered attribute details. Application shall save those information to reuse them for other connections. During normal connection, previously discovered device information can be reused.

For a normal connection, the response to this request is sent right away after saving the CTS, NDCS and RTUS content in the environment and registering TIPC in GATT to receive the notifications for the known attribute handle in CTS (Current Time) that would be notified. If the Client role is already enabled, a TIPC_ERROR_IND will be sent with the PRF_ERR_REQ_DISALLOWED

void app_tipc_rd_char_req (uint8_t *char_code*, uint16_t *conhdl*)

Parameters:

in	*char_code*	Code for which characteristic to read. ● TIPC_RD_CTS_CURR_TIME ● TIPC_RD_CTS_LOCAL_TIME_INFO ● TIPC_RD_CTS_REF_TIME_INFO ● TIPC_RD_CTS_CURR_TIME_CLI_CFG ● TIPC_RD_NDCS_TIME_WITH_DST

		• TIPC_RD_RTUS_TIME_UPD_STATE
in	*conhdl*	Connection handle for which the Time Client role is enabled.

Response:

- TIPC_CT_IND
- TIPC_CT_NTF_CFG_RD_RSP
- TIPC_LTI_RD_RSP
- TIPC_RTI_RD_RSP
- TIPC_TDST_RD_RSP
- TIPC_TUS_RD_RSP

Description:

This API is used by the application to request sending of a GATT_READ_CHAR_REQ with the parameters deduced from the char_code. Upon reception of this message, TIPC checks whether the parameters are correct, then if the handle for the characteristic is valid (not 0x0000), the request is sent to GATT.

Parsing intelligence of the received response is added in this API handler in order to make easier the implementation of an application. When the peer has responded to GATT, and the response is routed to TIPC, either the TIPC_CT_IND or the TIPC_CT_NTF_CFG_RD_RSP or the TIPC_LTI_RD_RSP or the TIPC_RTI_RD_RSP or the TIPC_TDST_RD_RSP or the TIPC_TUS_RD_RSP message will be generically built following the last read request sent.

void app_tipc_ct_ntf_cfg_req (uint16_t *cfg_val*, uint16_t *conhdl*)

Parameters:

in	*cfg_val*	Stop/notify/indicate value to configure into the peer characteristic: • PRF_CLI_STOP_NTFIND • PRF_CLI_START_NTF • PRF_CLI_START_IND
in	*conhdl*	Connection handle for which the Time Client role is enabled.

Response:

TIPC_WR_CHAR_RSP or TIPC_ERROR_IND

Description:

This API is used by the application to send a GATT_WRITE_CHAR_REQ to the Current Time Client Configuration Characteristic Descriptor with the parameter cfg_val.

When the peer has responded to GATT, and the response is routed to TIPC, the TIPC_WR_CHAR_RSP message will be generically built and sent to Application. An error status is also possible either for the Write procedure or for the application request, in the second case, the TIPC_ERROR_IND message is sent to Application

void app_tipc_wr_time_udp_ctnl_pt_req (uint8_t *value*, uint16_t *conhdl*)

Parameters:

in	*value*	Time Update Control Point value to write: • TIPS_TIME_UPD_CTNL_PT_GET • TIPS_TIME_UPD_CTNL_PT_CANCEL
in	*conhdl*	Connection handle for which the Time Client role is enabled.

Response:

TIPC_WR_CHAR_RSP or TIPC_ERROR_IND

Description:

This API is used by the application to send a GATT_WRITE_CHAR_REQ to the Time Update Control Point Characteristic. Upon reception of this message, TIPC checks whether the parameters are correct, then if the handle for the characteristic is valid (not 0x0000) and

whether it is writable or not - if all OK, the request is sent to GATT, otherwise a TIPC_ERROR_IND message is build for the Application.

When the peer has responded to GATT, and the response is routed to TIPC, the TIPC_WR_CHAR_RSP message will be generically built and sent to the Application. An error status is also possible for the Write procedure, it will be sent through the same message.

3.11.2 Time Profile Client Task API

Detailed Description

Time Profile Client Task APIs are used to handle the message from TIPC or APP.

Data Structure Documentation

struct tipc_enable_cfm

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	status	status
struct tipc_cts_content	cts	Existing handle values cts.
struct tipc_ndcs_content	ndcs	Existing handle values ndcs.
struct tipc_rtus_content	rtus	Existing handle values rtus.

struct tipc_error_ind

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	status	Status.

struct tipc_wr_char_rsp

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	status	Status.

struct tipc_ct_ind

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	ind_type	Indication type.
struct tip_curr_time	ct_val	Current Time Value.

struct tipc_ct_ntf_cfg_rd_rsp

Data Fields:

uint16_t	conhdl	Connection handle.
uint16_t	ntf_cfg	Notification Configuration.

struct tipc_lti_rd_rsp

Data Fields:

uint16_t	conhdl	Connection handle.
struct	lti_val	Current Time Value.

tip_loc_time_info		

struct tipc_rti_rd_rsp

Data Fields:

uint16_t	conhdl	Connection handle.
struct tip_ref_time_info	rti_val	Current Time Value.

struct tipc_tdst_rd_rsp

Data Fields:

uint16_t	conhdl	Connection handle.
struct tip_time_with_dst	tdst_val	Current Time Value.

struct tipc_tus_rd_rsp

Data Fields:

uint16_t	conhdl	Connection handle.
struct tip_time_upd_state	tus_val	Current Time Value.

Function Documentation

int app_tipc_enable_cfm_handler (ke_msg_id_t const *msgid*, struct tipc_enable_cfm * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	TIPC_ENABLE_CFM
in	*param*	Pointer to struct tipc_enable_cfm
in	*dest_id*	TASK_APP
in	*src_id*	TASK_TIPC

Returns:

If the message was consumed or not.

Description:

This API is used by the Client to either send the discovery results of CTS, NDCS or RTUS and confirm enabling of the Client role (status = PRF_ERR_OK), or to simply confirm enabling of Client role if it is a normal connection and the attribute details are already known (status = PRF_ERR_OK), or to inform the application that the discovery process has been stopped because of a missing attribute (status = PRF_ERR_STOP_DISC_CHAR_MISSING).

int app_tipc_error_ind_handler (ke_msg_id_t const *msgid*, struct tipc_error_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	TIPC_ERROR_IND
in	*param*	Pointer to struct tipc_error_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_TIPC

Returns:

If the message was consumed or not.

Description:

This API is used by the Client role to inform the Application of an error occurred in different situations. The error codes are proprietary and defined in prf_types.h. An error may occur during attribute discovery or due to application request parameters. Following reception of this message, the application will decide the necessary action.

int app_tipc_wr_char_rsp_handler (ke_msg_id_t const *msgid*, struct tipc_wr_char_rsp * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	TIPC_WR_CHAR_RSP
in	*param*	Pointer to struct tipc_wr_char_rsp
in	*dest_id*	TASK_APP
in	*src_id*	TASK_TIPC

Returns:

If the message was consumed or not.

Description:

This API is used by the Client role to inform the Application of a received write response. The status and the data from the write response are passed directly to Application, which must interpret them based on the request it made.

int app_tipc_ct_ind_handler (ke_msg_id_t const *msgid*, struct tipc_ct_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	TIPC_CT_IND
in	*param*	Pointer to struct tipc_ct_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_TIPC

Returns:

If the message was consumed or not.

Description:

This API is used by the Client role to inform the Application of a received current time value. The ind_type parameter informs the application if the value has been notified by the Time Client or if it has been received as a read response.

int app_tipc_ct_ntf_cfg_rd_rsp_handler (ke_msg_id_t const *msgid*, struct tipc_ct_ntf_cfg_rd_rsp * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	TIPC_CT_NTF_CFG_RD_RSP
in	*param*	Pointer to struct tipc_ct_ntf_cfg_rd_rsp
in	*dest_id*	TASK_APP
in	*src_id*	TASK_TIPC

Returns:

If the message was consumed or not.

Description:

This API is used by the Client role to inform the application that the notification configuration value for the Current Time characteristic has been successfully read and to provide this value.

int app_tipc_lti_rd_rsp_handler (ke_msg_id_t const *msgid*, struct tipc_lti_rd_rsp * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	TIPC_LTI_RD_RSP
in	*param*	Pointer to struct tipc_lti_rd_rsp
in	*dest_id*	TASK_APP
in	*src_id*	TASK_TIPC

Returns:

If the message was consumed or not.

Note:

Time Zone Characteristic UUID: 0x2A0E Min value : -48 (UTC-12:00), Max value : 56 (UTC+14:00) -128 : Time zone offset is not known

DST Offset Characteristic UUID: 0x2A2D Min value : 0, Max value : 8 255 = DST is not known

Description:

This API is used by the Client role to inform the application that the LTI characteristic value has been successfully read and to provide this value.

int app_tipc_rti_rd_rsp_handler (ke_msg_id_t const *msgid*, struct tipc_rti_rd_rsp * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	TIPC_RTI_RD_RSP
in	*param*	Pointer to struct tipc_rti_rd_rsp
in	*dest_id*	TASK_APP
in	*src_id*	TASK_TIPC

Returns:

If the message was consumed or not.

Note:

Time Source Characteristic UUID: 0x2A13 Min value : 0, Max value : 6 0 = Unknown 1 = Network Time Protocol 2 = GPS 3 = Radio Time Signal 4 = Manual 5 = Atomic Clock 6 = Cellular Network

Time Accuracy Characteristic UUID:0x2A12 Accuracy (drift) of time information in steps of 1/8 of a second (125ms) compared to a reference time source. Valid range is from 0 to 253 (0s to 31.5s). A value of 254 means Accuracy is out of range (> 31.5s). A value of 255 means Accuracy is unknown.

Days since last update about Reference Source Min value: 0, Max value: 254 255 = 255 or more days

Hours since update about Reference Source Min value: 0, Mac value: 23 255 = 255 or more days (If Days Since Update = 255, then Hours Since Update shall also be set to 255)

Description:

This API is used by the Client role to inform the application that the RTI characteristic value has been successfully read and to provide this value.

int app_tipc_tdst_rd_rsp_handler (ke_msg_id_t const *msgid*, struct tipc_tdst_rd_rsp * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	TIPC_TDST_RD_RSP
in	*param*	Pointer struct tipc_tdst_rd_rsp
in	*dest_id*	TASK_APP
in	*src_id*	TASK_TIPC

Returns:

If the message was consumed or not.

Note:

Time With DST Characteristic Structure - UUID: 0x2A11

Description:

This API message is used by the Client role to inform the application that the TDST characteristic value has been successfully read and to provide this value.

int app_tipc_tus_rd_rsp_handler (ke_msg_id_t const *msgid*, struct tipc_tus_rd_rsp * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	TIPC_TUS_RD_RSP
in	*param*	Pointer to struct tipc_tus_rd_rsp
in	*dest_id*	TASK_APP
in	*src_id*	TASK_TIPC

Returns:

If the message was consumed or not.

Note:

Time Update State Characteristic Structure - UUID: 0x2A17

The Time Update Status Characteristic exposes the status of the time update process and the result of the last update in the server.

Current State Min value: 0, Max value = 1 0 = Idle 1 = Update Pending

Result Min value: 0, Max Value : 5 0 = Successful 1 = Canceled 2 = No Connection To Reference 3 = Reference responded with an error 4 = Timeout 5 = Update not attempted after reset

Description:

This API is used by the Client role to inform the application that the TUS characteristic value has been successfully read and to provide this value.

3.11.3 Time Profile Server

Detailed Description

The Bluetooth Low Energy Time profile enables the user to obtain the current date and time, and related information such as time zone as exposed by the Current Time service in the peer device. Information of when next change of daylight savings time (DST) will occur can be retrieved from the peer exposed by the Next DST Change service. This profile also enables a device to request updating the time on the peer device as exposed by the Reference Time Update Service. Within the profile, two roles can be supported: Client and Server. The Time Server shall be a server. The Time Client shall be a client. The role of server is meant to be activated on the device that acts as Time Server and sends time values to the Client.

Function Documentation

void app_tips_create_db (uint8_t *features*)

Parameters:

in	*features*	Indicate if optional features are supported or not. Possible bit-mask values are: ● TIPS_CTS_LOC_TIME_INFO_SUP ● TIPS_CTS_REF_TIME_INFO_SUP ● TIPS_NDCS_SUP ● TIPS_RTUS_SUP

Response:

TIPS_CREATE_DB_CFM

Description:

This function shall be used to add one instance of the Current Time Service, optionally one instance of the Next DST Change Service and one instance of the Reference Time Update Service.

void app_tips_enable_req (uint16_t *conhdl*, uint8_t *sec_lvl*, uint8_t *con_type*, uint16_t *current_time_ntf_en*)

Parameters:

in	*conhdl*	Connection handle for which the Time Server role is enabled.
in	*sec_lvl*	Security level required for protection of attributes Service Hide and Disable are not permitted. Possible values are: ● PERM_RIGHT_ENABLE ● PERM_RIGHT_UNAUTH ● PERM_RIGHT_AUTH
in	*con_type*	Connection type: PRF_CON_DISCOVERY (0x00) , PRF_CON_NORMAL (0x01)
in	*current_time_ntf_en*	Value stored for Current Time Notification Client Configuration Char.

Response:

None or TIPS_ERROR_IND

Description:

This function is used for enabling the Time Server role of the Time profile.

void app_tips_upd_curr_time_req (uint16_t *conhdl*, struct tip_curr_time * *current_time*, uint8_t *enable_ntf_send*)

Parameters:

in	*conhdl*	Connection handle for which the Time Server role is enabled.
in	*current_time*	Pointer to the struct tip_curr_time containing Current Time value
in	*enable_ntf_send*	Define if a notification of new current time value will be send, 0: Disable, 1: Enable The enable_ntf_send parameter shall be used to be conform with the following CTS Specification requirement: If the time of the Current Time Server is changed because of reference time update, then no notification shall be sent to Current Time Service Client within the 15 minutes from the last notification, unless one of both of the two statements below are true: ● The new time information differs by more than 1 minute from the Current Time Server time previous to the update. ● The update was caused by the client (interacting with another service).

Response:

None or TIPS_ERROR_IND_SEND

Description:

This function is used by the application for requesting an update of the Current Time characteristic value.

void app_tips_upd_local_time_info_req (uint16_t *conhdl*, struct tip_loc_time_info * *loc_time_info*)

Parameters:

in	*conhdl*	Connection handle for which the Time Server role is enabled
in	*loc_time_info*	Pointer to the struct tip_loc_time_info containing Local Time Information

Response:

None or TIPS_ERROR_IND_SEND

Description:

This function is used by the application for updating the Local Time Information Characteristic value.

void app_tips_upd_ref_time_info_req (uint16_t *conhdl*, struct tip_ref_time_info * *ref_time_info*)

Parameters:

in	*conhdl*	Connection handle for which the Time Server role is enabled
in	*ref_time_info*	Pointer to the struct tip_ref_time_info containing Reference Time Information

Response:

None or TIPS_ERROR_IND_SEND

Description:

This function is used by the application for updating the Reference Time Information Characteristic value.

void app_tips_upd_time_dst_req (uint16_t *conhdl*, struct tip_time_with_dst * *time_with_dst*)

Parameters:

in	*conhdl*	Connection handle for which the Time Server role is enabled
in	*time_with_dst*	Pointer to the struct tip_time_with_dst containing Time With DST

Response:

None or TIPS_ERROR_IND_SEND

Description:

This function is used by the application for updating the Reference Time Information Characteristic value.

void app_tips_upd_time_upd_state_req (uint16_t *conhdl*, struct tip_time_upd_state * *time_upd_state*)

Parameters:

in	*conhdl*	Connection handle for which the Time Server role is enabled
in	*time_upd_state*	Pointer to the struct tip_time_upd_state containing Time Update State

Response:

None or TIPS_ERROR_IND_SEND

Description:

This function is used by the application for updating the Reference Time Information Characteristic value

3.11.4 Time Profile Profile Server Task API

Detailed Description

Time Profile Profile Server Task APIs are used to handle the message from TIPS or APP.

Data Structure Documentation

struct tips_create_db_cfm

Data Fields:

uint8_t	status	Status about database creation.

struct tips_disable_ind

Data Fields:

uint16_t	conhdl	COnnection Handle.
uint16_t	current_time_ntf_en	Current Time notification configuration.

struct tips_current_time_ccc_ind

Data Fields:

uint16_t	conhdl	Connection handle.
uint16_t	cfg_val	Configuration Value.

struct tips_time_upd_ctnl_pt_ind

Data Fields:

uint16_t	conhdl	Connection handle.
tip_time_upd_cont r_pt	value	Time Update Control Point value written by the peer client.

struct tips_enable_cfm

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	status	Status.

Function Documentation

int app_tips_create_db_cfm_handler (ke_msg_id_t const *msgid*, struct tips_create_db_cfm * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	TIPS_CREATE_DB_CFM
in	*param*	Pointer to the struct tips_create_db_cfm
in	*dest_id*	TASK_APP
in	*src_id*	TASK_TIPS

Returns:

If the message was consumed or not.

Description:

This handler shall be called after reception of a database creation. The status parameter indicates if requested services have been successfully added into the database or not.

int app_tips_disable_ind_handler (ke_msg_id_t const *msgid*, struct tips_disable_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	TIPS_DISABLE_IND
in	*param*	Pointer to the struct tips_disable_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_TIPS

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the application of a correct disable. The current notification configuration value for the Current Time characteristic is included in the parameters so that the higher application may safely keep the configuration until the next time the profile role is enabled.

int app_tips_error_ind_handler (ke_msg_id_t const *msgid*, struct prf_server_error_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	TIPS_ERROR_IND
in	*param*	Pointer to the struct prf_server_error_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_TIPS

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the Application of an occurred error.

int app_tips_current_time_ccc_ind_handler (ke_msg_id_t const *msgid*, struct tips_current_time_ccc_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	TIPS_CURRENT_TIME_CCC_IND
in	*param*	Pointer to the struct tips_current_time_ccc_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_TIPS

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the application about a modification of the Current Time Client Configuration characteristic value.

int app_tips_time_upd_ctnl_pt_ind_handler (ke_msg_id_t const *msgid*, struct tips_time_upd_ctnl_pt_ind * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	TIPS_TIME_UPD_CTNL_PT_IND
in	*param*	Pointer to the struct tips_time_upd_ctnl_pt_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_TIPS

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the application about a modification of the Time Update Control Point Characteristic value.

4. QN9020 BLE Protocol Stack

QN9020 supports complete BLE protocol stack including all layers from the Physical Layer to GAP and GATT layers. Any profiles and applications that are being used sit on top of the GAP and GATT layers of the stack. The APIs are provided for GAP, GATT and SM layers to help interact directly with the application and profiles.

4.1 Generic Access Profile (GAP)

4.1.1 Generic Access Profile API

Detailed Description

The GAP layer of the BLE Protocol Stack is responsible for handling the device's access modes and procedures, including device discovery, link establishment, link termination, initiation of security features, and device configuration.

Function Documentation

void app_gap_set_devname_req (uint8_t const * *name*, uint8_t *len*)

Parameters:

in	*name*	Name of the device to set
in	*len*	length for name

Response:

GAP_SET_DEVNAME_REQ_CMP_EVT

Description:

This function is used to set the device name as seen by remote device.

void app_gap_set_sec_req (uint8_t *sec_lvl*)

Parameters:

in	*sec_lvl*	Security settings to write, possible values are: ● GAP_NO_SEC ● GAP_SEC1_NOAUTH_PAIR_ENC ● GAP_SEC1_AUTH_PAIR_ENC ● GAP_SEC2_NOAUTH_DATA_SGN ● GAP_SEC2_AUTH_DATA_SGN

Response:

GAP_SET_SEC_REQ_CMP_EVT

Description:

This function is used to set security level of the device. It is advisable to set the security level as soon as the device starts.

void app_gap_read_ver_req (void)

Response:

GAP_READ_VER_REQ_CMP_EVT

Description:

This function is used to read the version information of the BLE stack.

void app_gap_read_bdaddr_req (void)

Response:

GAP_READ_BDADDR_REQ_CMP_EVT

Description:

This function is used to read the Bluetooth Address of the device.

void app_gap_dev_inq_req (uint8_t *inq_type*, uint8_t *own_addr_type*)

Parameters:

in	*inq_type*	Inquiry type, possible values are: ● GAP_GEN_INQ_TYPE ● GAP_LIM_INQ_TYPE ● GAP_KNOWN_DEV_INQ_TYPE
in	*own_addr_type*	the address type used when inquirying ● ADDR_PUBLIC ● ADDR_RAND

Response:

GAP_DEV_INQ_REQ_CMP_EVT GAP_DEV_INQ_RESULT_EVT
GAP_KNOWN_DEV_DISC_RESULT_EVT

Description:

This function is used to search devices within range.

void app_gap_dev_inq_cancel_req (void)

Response:

GAP_SCAN_REQ_CMP_EVT

Description:

This function is used to stop the current inquiry.

void app_gap_name_req (struct bd_addr * *p_addr*, uint8_t *addr_type*, uint8_t *own_addr_type*)

Parameters:

in	*p_addr*	Pointer to device address of peer
in	*addr_type*	Device address type of peer, possible values are: ● ADDR_PUBLIC ● ADDR_RAND
in	*own_addr_type*	Own address type, possible values are: ● ADDR_PUBLIC ● ADDR_RAND

Response:

GAP_NAME_REQ_CMP_EVT

Description:

This function is used to find out the user friendly name of peer device.

void app_gap_bond_req (struct bd_addr * *addr*, uint8_t *oob*, uint8_t *auth*, uint8_t *iocap*)

Parameters:

in	*addr*	Pointer to device address of peer
in	*oob*	Out-Of-Band present flag, possible values are: ● SMP_OOB_AUTH_DATA_NOT_PRESENT ● SMP_OOB_AUTH_DATA_FROM_REMOTE_DEV_PRESENT
in	*auth*	Authentication requirements, possible values are: ● SMP_AUTH_REQ_NO_MITM_NO_BOND ● SMP_AUTH_REQ_NO_MITM_BOND ● SMP_AUTH_REQ_MITM_NO_BOND ● SMP_AUTH_REQ_MITM_BOND
in	*iocap*	Input and output capabilities of local device, possible values are: ● SMP_IO_CAP_DISPLAY_ONLY ● SMP_IO_CAP_DISPLAY_YES_NO ● SMP_IO_CAP_KB_ONLY ● SMP_IO_CAP_NO_INPUT_NO_OUTPUT ● SMP_IO_CAP_KB_DISPLAY

GAP_BOND_REQ_CMP_EVT

Description:

This function is used to initiate bonding procedure.

void app_gap_bond_resp (uint16_t *conhdl*, uint8_t *reject*, uint8_t *oob*, uint8_t *auth*, uint8_t *iocap*)

Parameters:

in	*conhdl*	Connection handle
in	*reject*	Decision to accept or reject the bond request. 0x00: accept 0x01:

		reject
in	*oob*	Out-Of-Band present flag, possible values are: ● SMP_OOB_AUTH_DATA_NOT_PRESENT ● SMP_OOB_AUTH_DATA_FROM_REMOTE_DEV_PRESENT
in	*auth*	Authentication requirements, possible values are: ● SMP_AUTH_REQ_NO_MITM_NO_BOND ● SMP_AUTH_REQ_NO_MITM_BOND ● SMP_AUTH_REQ_MITM_NO_BOND ● SMP_AUTH_REQ_MITM_BOND
in	*iocap*	Input and output capabilities of local device, possible values are: ● SMP_IO_CAP_DISPLAY_ONLY ● SMP_IO_CAP_DISPLAY_YES_NO ● SMP_IO_CAP_KB_ONLY ● SMP_IO_CAP_NO_INPUT_NO_OUTPUT ● SMP_IO_CAP_KB_DISPLAY

Response:

None

Description:

This function is used to answer to bond request from peer device.

void app_gap_unpair_req (struct bd_addr * *addr*, uint8_t *nb_bond*)

Parameters:

in	*addr*	Desired BD address to be removed in the bond list
in	*nb_bond*	Number of bonded devices

Response:

None

Description:

This function is used to update the stack's bonding information.

void app_gap_security_req (struct bd_addr * *addr*)

Parameters:

in	*addr*	Desired BD address to be removed in the bond list

Response:

GAP_BOND_REQ_CMP_EVT or SMPC_SEC_STARTED_IND

Description:

This function is used to initiate a encryption or pairing procedure. For a unbonded device, this will initiate pairing, or this will initiate encryption.

void app_gap_le_create_conn_req (struct bd_addr * *addr*, uint8_t *addr_type*, uint8_t *own_addr_type*, uint16_t *conn_intv_min*, uint16_t *conn_intv_max*, uint16_t *cnnn_timeout*)

Parameters:

in	*addr*	The address of the remote device to which the connection will be created
in	*addr_type*	The address type of the remote device, possible values are: ● ADDR_PUBLIC ● ADDR_RAND
in	*own_addr_type*	Own address type, possible values are: ● ADDR_PUBLIC ● ADDR_RAND
in	*conn_intv_min*	Minimum of connection interval
in	*conn_intv_max*	Maximum of connection interval
in	*cnnn_timeout*	Link supervision timeout

Response:

GAP_LE_CREATE_CONN_REQ_CMP_EVT

Description:

This function is used to create a Link Layer connection to a connectable device. This is initiated by central device, which will become the master of the link.

void app_gap_le_cancel_conn_req (void)

Response:

GAP_CANCEL_CONN_REQ_CMP_EVT

Description:

This function is used to cancel an existing connection request. This function shall only be called after the GAP_LE_CREATE_CONN_REQ message has been issued and before GAP_LE_CREATE_CONN_REQ_CMP_EVT message.

void app_gap_discon_req (uint16_t *conhdl*)

Parameters:

in	*conhdl*	The connection handle of the connection is to be disconnected.

Response:

GAP_DISCON_CMP_EVT

Description:

This function is used to disconnect an existing BLE connection.

void app_gap_set_bondable_mode_req ()

Response:

GAP_SET_MODE_REQ_CMP_EVT

Description:

This function is used to set the device to bondable mode.

void app_gap_adv_start_req (uint16_t *mode*, uint8_t * *adv_data*, uint8_t *adv_data_len*, uint8_t * *scan_rsp_data*, uint8_t *scan_rsp_data_len*, uint16_t *adv_intv_min*, uint16_t *adv_intv_max*)

Parameters:

in	*mode*	Device mode to set, possible values are: ● GAP_NON_DISCOVERABLE ● GAP_GEN_DISCOVERABLE ● GAP_LIM_DISCOVERABLE ● GAP_NON_CONNECTABLE ● GAP_UND_CONNECTABLE ● GAP_DIR_CONNECTABLE
in	*adv_data*	Pointer to advertising data used in the advertising packets
in	*adv_data_len*	The length of advertising data
in	*scan_rsp_data*	Pointer to Scan Response data used in the advertising packets
in	*scan_rsp_data_len*	The length of Scan Response data
in	*adv_intv_min*	Minimum interval for advertising
in	*adv_intv_max*	Maximum interval for advertising

Response:

GAP_SET_MODE_REQ_CMP_EVT

Description:

This function is used to set the device to advertising.

Note:

The stack will keep advertising with new parameters if calling this function in advertising state. The adv_intv_min and adv_intv_max shall not be set to less than 0x00A0(100 ms) if the mode is GAP_NON_DISCOVERABLE.

void app_gap_adv_stop_req (void)

Response:

GAP_ADV_REQ_CMP_EVT

Description:

This function is used to stop advertising.

void app_gap_le_rd_wlst_size_req (void)

Response:

GAP_LE_RD_WLST_SIZE_CMD_CMP_EVT

Description:

This function is used to read the total number of white list entries that can be stored in the BLE chip.

void app_gap_le_add_dev_to_wlst_req (uint8_t *addr_type*, struct bd_addr * *addr*)

Parameters:

in	*addr_type*	The address type of the stored device, possible values are: ● ADDR_PUBLIC ● ADDR_RAND
in	*addr*	Pointer to device address to be stored

Response:

GAP_LE_ADD_DEV_TO_WLST_REQ_CMP_EVT

Description:

This function is used to add a single device to the white list stored in the BLE chip.

void app_gap_le_rmv_dev_frm_wlst_req (bool *all_dev*, uint8_t *addr_type*, struct bd_addr * *addr*)

Parameters:

in	*all_dev*	Flag to determine if all the devices will be removed from the white list or only the one specified in the structure
in	*addr_type*	The address type of the removed device, possible values are: ● ADDR_PUBLIC ● ADDR_RAND
in	*addr*	Pointer to device address to be removed

Response:

GAP_LE_RMV_DEV_FRM_WLST_REQ_CMP_EVT

Description:

This function is used to remove device from white list stored in the BLE chip.

void app_gap_le_rd_remote_feat_req (uint16_t *conhdl*)

Parameters:

in	*conhdl*	The connection handle of the connection is to be read

Response:

GAP_LE_RD_REMOTE_FEAT_REQ_CMP_EVT

Description:

This function is used to request a list of the used LE features from the remote device.

void app_gap_rd_rem_ver_info_req (uint16_t *conhdl*)

Parameters:

in	*conhdl*	Specifies which Connection Handles version information to get

Response:

GAP_RD_REM_VER_INFO_CMP_EVT

Description:

This function is used to obtain the values for the version information for the remote device.

void app_gap_set_random_addr_req (struct bd_addr * *addr*)

Parameters:

in	*addr*	Pointer to device address to be set

Response:

GAP_SET_RANDOM_ADDR_REQ_CMP_EVT

Description:

This function is used to set the Random Device Address in the BLE chip.

void app_gap_param_update_req (uint16_t *conhdl*, struct gap_conn_param_update * *conn_par*)

Parameters:

in	*conhdl*	Connection handle to be used to identify a connection
in	*conn_par*	Pointer to the stuct gap_conn_param_update containing connection parameters

Response:

GAP_PARAM_UPDATE_RESP
GAP_CHANGE_PARAM_REQ_CMP_EVT

Description:

This function is used to change a set of new connection parameters. The peripheral is the only one that can send this request.

void app_gap_change_param_req (uint16_t *conhdl*, uint16_t *result*, struct gap_conn_param_update * *conn_par*)

Parameters:

in	*conhdl*	Connection handle to be used to identify a connection
in	*result*	Result of the connection parameters request, 0x0000: accept 0x0001: reject
in	*conn_par*	Pointer to the stuct gap_conn_param_update containing connection parameters

Response:

GAP_CHANGE_PARAM_REQ_CMP_EVT

Description:

This function is used to send parameters update change by master.

Note:

This function is called in two occasions: 1. Message GAP_PARAM_UPDATE_REQ_IND will be a hint that slave requests connection parameters update. At this time, the parameter of result is meaningful. 2. Device(master) wants to change the current connection parameters.

void app_gap_reset_req (void)

Response:

GAP_RESET_REQ_CMP_EVT

Description:

This function is used to reset the device and the BLE stack will be initialized.

void app_gap_set_recon_addr_req (uint16_t *conhdl*, uint16_t *attrhdl*)

Parameters:

in	*conhdl*	Connection handle to be used to identify a connection

in	*attrhdl*	Attribute handle of the reconnection address

Response:

GAP_SET_RECON_ADDR_REQ_CMP_EVT

Description:

This function is used to set the reconnection address attribute of the peripheral device by the central device.

void app_gap_set_ph_privacy_req (uint16_t *enable*, uint16_t *conhdl*, uint16_t *attrhdl*)

Parameters:

in	*enable*	Specify to enable or disable the privacy flag in the attribute
in	*conhdl*	Connection handle to be used to identify a connection
in	*attrhdl*	Attribute handle of the reconnection address

Response:

GAP_SET_PH_PRIVACY_REQ_CMP_EVT

Description:

This function is used to set the privacy settings of the peer peripheral device.

void app_gap_set_privacy_req (uint8_t *priv_flag*, uint8_t *recon_addr_visible*, uint8_t *set_to_ll*)

Parameters:

in	*priv_flag*	privacy flag, possible values are: ● DEV_PRIV_DIS ● CT_PRIV_EN ● PH_PRIV_EN ● BCST_PRIV_EN ● OBS_PRIV_EN ● OBS_PRIV_RESOLVE
in	*recon_addr_visible*	Reconnection address visible flag, valid only for Peripheral ● 0x00 Hide reconnection address ● 0x01 Show/Expose reconnection address
in	*set_to_ll*	Flag to set the generated random address to link layer, true or false

Response:

GAP_SET_PRIVACY_REQ_CMP_EVT

Description:

This function is used to enable privacy feature of the local device.

void app_gap_channel_map_req (bool *update_map*, uint16_t *conhdl*, struct le_chnl_map * *chmap*)

Parameters:

in	*update_map*	Flag to either read the map or update it, false:read, true: update
in	*conhdl*	Connection handle to be used to read a channel map
in	*chmap*	Pointer to the struct le_chnl_map value which is used to update

Response:

GAP_CHANNEL_MAP_CMP_EVT

Description:

This function is used to read the current channel map or change it with the new channel map in the command parameter.

Note:

The Channel Map shall only be updated when the local device supports the Master role.

void app_gap_read_rssi_req (uint16_t *conhdl*)

Parameters:

in	*conhdl*	The Handle for the connection for which the RSSI is to be read

Response:

GAP_READ_RSSI_REQ_CMP_EVT

Description:

This function is used to read Received Signal Strength Indication (RSSI) value.

4.1.2 Generic Access Profile Task API

Detailed Description

GAP Task APIs are used to handle the message from GAP or APP.

Data Structure Documentation

struct gap_conn_param_update

Data Fields:

uint16_t	intv_min	Connection interval minimum.
uint16_t	intv_max	Connection interval maximum.
uint16_t	latency	Latency.
uint16_t	time_out	Supervision timeout.

struct gap_link_info

Data Fields:

uint8_t	status	Confirmation status.
uint16_t	conhdl	Connection handle.
uint8_t	peer_addr_type	Peer address type.
struct bd_addr	peer_addr	Peer BT address.
uint16_t	con_interval	Connection interval.
uint16_t	con_latency	Connection latency.
uint16_t	sup_to	Link supervision timeout.
uint8_t	clk_accuracy	Clock accuracy.

struct gap_rd_wlst_size_cmd_complete

Data Fields:

uint8_t	status	Status of the command reception.
uint8_t	wlst_size	White List size.

struct gap_le_create_conn_req_cmp_evt

Data Fields:

struct gap_link_info	conn_info	Connection establishment information.

struct gap_bond_req_ind

Data Fields:

struct bd_addr	addr	Device BD Address.
uint8_t	index	Device record index.
uint8_t	auth_req	Authentication Requirements from peer.
uint8_t	io_cap	IO capabilities.
uint8_t	oob_data_flg	Out Of Band Data presence flag.
uint8_t	max_enc_size	Maximum Encryption Key Size.
uint8_t	ikey_dist	Initiator Key Distribution.
uint8_t	rkey_dist	Responder Key Distribution.

struct gap_discon_cmp_evt

Data Fields:

uint8_t	reason	Reason.
uint8_t	status	Status.
uint16_t	conhdl	Connection handle.

struct gap_rd_rem_ver_info_cmp_evt

Data Fields:

uint8_t	status	Status for command reception.
uint16_t	conhdl	Connection handle.
uint8_t	vers	LMP version.
uint16_t	compid	Manufacturer name.
uint16_t	subvers	LMP subversion.

struct gap_set_recon_addr_req_cmp_evt

Data Fields:

uint8_t	status	Status.

struct gap_set_ph_privacy_req_cmp_evt

Data Fields:

uint8_t	status	Status.

struct gap_set_privacy_req_cmp_evt

Data Fields:

uint8_t	status	Status.

struct gap_param_update_resp

Data Fields:

uint8_t	status	status
uint16_t	result	Result.
uint16_t	conhdl	Connection handle.

struct gap_param_update_req_ind

Data Fields:

uint16_t	conhdl	Connection handle.
struct gap_conn_param_update	conn_param	Connection parameter update request.

struct gap_event_common_cmd_complete

Data Fields:

uint8_t	status	Command complete status.

struct gap_dev_inq_result_evt

Data Fields:

uint8_t	nb_resp	Number of responses.
struct adv_report	adv_rep	advertising report

struct gap_name_req_cmp_evt

Data Fields:

uint8_t	status	status of the name request
struct bd_name	bdname	Characteristic name.

struct gap_bond_req_cmp_evt

Data Fields:

uint16_t	conhdl	connection handle
uint8_t	idx	record index
uint8_t	status	status
uint8_t	key_size	Key size for the LTK/STK agreed upon during pairing features exchange)
uint8_t	sec_prop	Security properties of the keys.
uint8_t	bonded	Bonded status.

struct gap_le_rd_remote_feat_req_cmp_evt

Data Fields:

uint8_t	status	Status of read remote feature request command.
uint16_t	conhdl	Connection handle.
struct le_features	feats_used	LE Features used.

struct gap_reset_req_cmp_evt

Data Fields:

uint8_t	status	Status of the reset command.

struct gap_change_param_req_cmp_evt

Data Fields:

uint8_t	status	Status of the change parameter request.
uint16_t	con_interval	Connection interval value.
uint16_t	con_latency	Connection latency value.
uint16_t	sup_to	Supervision timeout.

struct gap_set_devname_req_cmp_evt

Data Fields:

uint8_t	status	status

struct gap_read_ver_req_cmp_evt

Data Fields:

uint8_t	status	Status.
uint8_t	hci_ver	HCI version.
uint8_t	lmp_ver	LMP version.
uint8_t	host_ver	Host version.
uint16_t	hci_subver	HCI revision.
uint16_t	lmp_subver	LMP subversion.
uint16_t	host_subver	Host revision.
uint16_t	manuf_name	Manufacturer name.

struct gap_set_sec_req_cmp_evt

Data Fields:

uint8_t	status	status
uint8_t	sec	security mode

struct gap_set_random_addr_req_cmp_evt

Data Fields:

uint8_t	status	Status.
struct bd_addr	addr	Device BD Address.

struct gap_read_bdaddr_req_cmp_evt

Data Fields:

uint8_t	status	status
struct bd_addr	addr	Device BD Address.

struct gap_channel_map_cmp_evt

Data Fields:

uint16_t	conhdl	Connection handle.
uint8_t	status	Status.
struct le_chnl_map	chmap	Channel map.

struct gap_read_rssi_req_cmp_evt

Data Fields:

uint8_t	status	Status.
uint8_t	rssi	RSSI value.

Function Documentation

int app_gap_ready_evt_handler (ke_msg_id_t const *msgid*, void const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GAP_READY_EVT
in	*param*	NULL pointer
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GAP

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the application that the BLE stack initialization has completed.

int app_gap_reset_req_cmp_handler (ke_msg_id_t const *msgid*, struct gap_reset_req_cmp_evt const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GAP_RESET_REQ_CMP_EVT
in	*param*	Pointer to struct gap_reset_req_cmp_evt
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GAP

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the application that device reset has completed.

int app_gap_set_devname_req_cmp_evt_handler (ke_msg_id_t const *msgid*, struct gap_event_common_cmd_complete const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GAP_SET_DEVNAME_REQ_CMP_EVT
in	*param*	Pointer to struct gap_event_common_cmd_complete

in	*dest_id*	TASK_APP
in	*src_id*	TASK_GAP

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the application that the set device name request has completed.

int app_gap_set_sec_req_cmp_evt_handler (ke_msg_id_t const *msgid*, struct gap_set_sec_req_cmp_evt const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GAP_SET_SEC_REQ_CMP_EVT
in	*param*	Pointer to struct gap_set_sec_req_cmp_evt
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GAP

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the application that the set security mode has completed.

int app_gap_read_ver_req_cmp_evt_handler (ke_msg_id_t const *msgid*, struct gap_read_ver_req_cmp_evt const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GAP_READ_VER_REQ_CMP_EVT
in	*param*	Pointer to struct gap_read_ver_req_cmp_evt
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GAP

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the application that the read the version information of the stack has completed.

int app_gap_read_bdaddr_req_cmp_evt_handler (ke_msg_id_t const *msgid*, struct gap_read_bdaddr_req_cmp_evt const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GAP_READ_BDADDR_REQ_CMP_EVT
in	*param*	Pointer to struct gap_read_bdaddr_req_cmp_evt
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GAP

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the application that the read the Bluetooth Address of the device has completed.

int app_gap_dev_inq_result_handler (ke_msg_id_t const *msgid*, struct gap_dev_inq_result_evt const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GAP_DEV_INQ_RESULT_EVT

in	*param*	Pointer to struct gap_dev_inq_result_evt
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GAP

Returns:

If the message was consumed or not.

Description:

This handler is used to indicate that a BLE device has responded so far during inquiry process.

int app_gap_dev_inq_cmp_handler (ke_msg_id_t const *msgid*, struct gap_event_common_cmd_complete const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GAP_DEV_INQ_REQ_CMP_EVT
in	*param*	Pointer to struct gap_event_common_cmd_complete
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GAP

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the application that the inquiry process has completed.

int app_gap_scan_req_cmp_evt_handler (ke_msg_id_t const *msgid*, struct gap_event_common_cmd_complete const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GAP_SCAN_REQ_CMP_EVT
in	*param*	Pointer to struct gap_event_common_cmd_complete
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GAP

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the application that the scan command has completed.

int app_gap_set_mode_req_cmp_evt_handler (ke_msg_id_t const *msgid*, struct gap_event_common_cmd_complete const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GAP_SET_MODE_REQ_CMP_EVT
in	*param*	Pointer to struct gap_event_common_cmd_complete
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GAP

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the application that set mode command has completed.

int app_gap_adv_req_cmp_evt_handler (ke_msg_id_t const *msgid*, struct gap_event_common_cmd_complete const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GAP_ADV_REQ_CMP_EVT

in	*param*	Pointer to struct gap_event_common_cmd_complete
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GAP

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the application that stop advertising has completed.

int app_gap_le_create_conn_req_cmp_evt_handler (ke_msg_id_t const *msgid*, struct gap_le_create_conn_req_cmp_evt const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GAP_LE_CREATE_CONN_REQ_CMP_EVT
in	*param*	Pointer to struct gap_le_create_conn_req_cmp_evt
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GAP

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the application the outcome of connection establishment.

int app_gap_cancel_conn_req_cmp_evt_handler (ke_msg_id_t const *msgid*, void const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GAP_CANCEL_CONN_REQ_CMP_EVT
in	*param*	NULL
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GAP

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the application the outcome of cancel connection.

int app_gap_discon_cmp_evt_handler (ke_msg_id_t const *msgid*, struct gap_discon_cmp_evt const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GAP_DISCON_CMP_EVT
in	*param*	Pointer to struct gap_discon_cmp_evt
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GAP

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the application the BLE link has been disconnected.

int app_gap_name_req_cmp_handler (ke_msg_id_t const *msgid*, struct gap_name_req_cmp_evt const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GAP_NAME_REQ_CMP_EVT
in	*param*	Pointer to struct gap_name_req_cmp_evt
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GAP

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the application the name request has completed.

int app_gap_le_rd_wlst_size_cmd_cmp_handler (ke_msg_id_t const *msgid*, struct gap_rd_wlst_size_cmd_complete const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Handles read the white list size of the local device complete from the GAP.

Parameters:

in	*msgid*	GAP_LE_RD_WLST_SIZE_CMD_CMP_EVT
in	*param*	Pointer to struct gap_rd_wlst_size_cmd_complete
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GAP

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the application the read white list size command has completed.

int app_gap_le_add_dev_to_wlst_req_cmp_handler (ke_msg_id_t const *msgid*, struct gap_event_common_cmd_complete const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Handles add device to white list complete from the GAP.

Parameters:

in	*msgid*	GAP_LE_ADD_DEV_TO_WLST_REQ_CMP_EVT
in	*param*	Pointer to struct gap_event_common_cmd_complete
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GAP

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the application the outcome of add device to white list.

int app_gap_le_rmv_dev_from_wlst_req_cmp_handler (ke_msg_id_t const *msgid*, struct gap_event_common_cmd_complete const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Handles remove device from white list complete from the GAP.

Parameters:

in	*msgid*	GAP_LE_RMV_DEV_FRM_WLST_REQ_CMP_EVT
in	*param*	Pointer to struct gap_event_common_cmd_complete
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GAP

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the application the outcome of remove device from white list.

int app_gap_le_rd_remote_feat_req_cmp_handler (ke_msg_id_t const *msgid*, struct gap_le_rd_remote_feat_req_cmp_evt const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Handles read remote features complete from the GAP.

Parameters:

in	*msgid*	GAP_LE_RD_REMOTE_FEAT_REQ_CMP_EVT
in	*param*	Pointer to struct gap_le_rd_remote_feat_req_cmp_evt
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GAP

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the application that read remote features command has completed.

int app_gap_rd_remote_ver_info_cmp_handler (ke_msg_id_t const *msgid*, struct gap_rd_rem_ver_info_cmp_evt const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Handles read remote version information complete from the GAP.

Parameters:

in	*msgid*	GAP_RD_REM_VER_INFO_CMP_EVT
in	*param*	Pointer to struct gap_rd_rem_ver_info_cmp_evt
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GAP

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the application the version information of remote device.

int app_gap_set_random_add_req_cmp_handler (ke_msg_id_t const *msgid*, struct gap_set_random_addr_req_cmp_evt const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Handles set random address complete from the GAP.

Parameters:

in	*msgid*	GAP_SET_RANDOM_ADDR_REQ_CMP_EVT
in	*param*	Pointer to struct gap_set_random_addr_req_cmp_evt
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GAP

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the application that the set random address command has completed.

int app_gap_param_update_resp_handler (ke_msg_id_t const *msgid*, struct gap_param_update_resp const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Handles param update response from the GAP.

Parameters:

in	*msgid*	GAP_PARAM_UPDATE_RESP
in	*param*	Pointer to struct gap_param_update_resp
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GAP

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the application of the outcome of the connection parameter update by slave.

int app_gap_change_param_req_cmp_handler (ke_msg_id_t const *msgid*, struct gap_change_param_req_cmp_evt const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Handles change param request complete from the GAP.

Parameters:

in	*msgid*	GAP_CHANGE_PARAM_REQ_CMP_EVT
in	*param*	Pointer to struct gap_change_param_req_cmp_evt
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GAP

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the application of the outcome of change connection parameter.

int app_gap_set_recon_addr_req_cmp_handler (ke_msg_id_t const *msgid*, struct gap_set_recon_addr_req_cmp_evt const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Handles set reconnection address complete event from the GAP.

Parameters:

in	*msgid*	GAP_SET_RECON_ADDR_REQ_CMP_EVT
in	*param*	Pointer to struct gap_set_recon_addr_req_cmp_evt
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GAP

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the application of the outcome of set reconnection address.

int app_gap_set_ph_privacy_req_cmp_handler (ke_msg_id_t const *msgid*, struct gap_set_ph_privacy_req_cmp_evt const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Handles set the privacy settings of the peer peripheral device complete event from the GAP.

Parameters:

in	*msgid*	GAP_SET_PH_PRIVACY_REQ_CMP_EVT
in	*param*	Pointer to struct gap_set_ph_privacy_req_cmp_evt
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GAP

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the application of the outcome of enable or disable peer privacy.

int app_gap_set_privacy_req_cmp_handler (ke_msg_id_t const *msgid*, struct gap_set_privacy_req_cmp_evt const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Handles set privacy feature of the local device complete event from the GAP.

Parameters:

in	*msgid*	GAP_SET_PRIVACY_REQ_CMP_EVT
in	*param*	Pointer to struct gap_set_privacy_req_cmp_evt
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GAP

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the application of the outcome of enable or disable local privacy.

int app_gap_channel_map_cmp_handler (ke_msg_id_t const *msgid*, struct gap_channel_map_cmp_evt const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Handles channel map update operation complete event from the GAP.

Parameters:

in	*msgid*	GAP_CHANNEL_MAP_CMP_EVT
in	*param*	Pointer to struct gap_channel_map_cmp_evt
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GAP

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the application of the outcome of set or read channel map.

int app_gap_read_rssi_req_cmp_handler (ke_msg_id_t const *msgid*, struct gap_read_rssi_req_cmp_evt const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Handles read RSSI value complete event from the GAP.

Parameters:

in	*msgid*	GAP_READ_RSSI_REQ_CMP_EVT
in	*param*	Pointer to struct gap_read_rssi_req_cmp_evt
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GAP

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the application of the value of RSSI.

int app_gap_param_update_req_ind_handler (ke_msg_id_t const *msgid*, struct gap_param_update_req_ind const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GAP_PARAM_UPDATE_REQ_IND
in	*param*	Pointer to struct gap_param_update_req_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GAP

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the application of parameter update request indication from slave.

int app_gap_bond_req_cmp_ind_handler (ke_msg_id_t const *msgid*, struct gap_bond_req_cmp_evt const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GAP_BOND_REQ_CMP_EVT
in	*param*	Pointer to struct gap_bond_req_cmp_evt
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GAP

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the application of the outcome of bond request.

int app_gap_bond_req_ind_handler (ke_msg_id_t const *msgid*, struct gap_bond_req_ind const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GAP_BOND_REQ_IND
in	*param*	Pointer to struct gap_bond_req_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GAP

Returns:

If the message was consumed or not.

Description:

This handler is used to inform the application that remote devie wants to bond with our device. The application needs to send GAP_BOND_RESP to GAP block to indicate response to the bonding request.

int app_gatt_resource_access_req_handler (ke_msg_id_t const *msgid*, struct gatt_resource_access_req const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GATT_RESOURCE_ACCESS_REQ
in	*param*	Pointer to struct gatt_resource_access_req
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GATT

Returns:

If the message was consumed or not.

Description:

This handler is used to tell GATT app create database complete. When response is received by GATT, peer device is able to access database

4.2 Generic Attribute Profile (GATT)

4.2.1 Generic Attribute Profile API

Detailed Description

The Generic Attribute Profile (GATT) defines the service framework using the Attribute Protocol for discovering services and for reading and writing characteristic values on a peer device.

GATT APIs are used by APP to serch the service details which include UUID, start handle and end handle from the peer device.

Function Documentation

void app_gatt_disc_svc_req (uint8_t *req_type*, uint16_t *conhdl*)

Parameters:

in	*req_type*	GATT request type: ● GATT_DISC_ALL_SVC ● GATT_DISC_BY_UUID_SVC ● GATT_DISC_INCLUDED_SVC
in	*conhdl*	Connection handle.

Response:

GATT_DISC_SVC_ALL_CMP_EVT or GATT_DISC_SVC_BY_UUID_CMP_EVT or GATT_DISC_SVC_INCL_CMP_EVT and GATT_DISC_CMP_EVT

Description:

This API is used by the application to send a GATT_DISC_SVC_REQ with the parameters deduced from the req_type and desired_svc. The definition for the different codes for req_type can be found in gatt.h. Upon reception of this message, GATT will checks whether the parameters are correct, if not correct then the GATT_DISC_CMP_EVT message will be generically built and sent to Application directly. An error status is also possible be either GATT_INVALID_PARAM_ERR or GATT_INVALID_TYPE_IN_SVC_SEARCH. If parameter is correct, the GATT_DISC_SVC_ALL_CMP_EVT message will be received with the searched UUID, start handle and end handle together. Once all serviced be got, in the second case, the GATT_DISC_CMP_EVT message is sent to Application.

void app_gatt_disc_char_req (uint8_t *req_type*, uint16_t *conhdl*)

Parameters:

in	*req_type*	GATT request type: ● GATT_DISC_ALL_CHAR ● GATT_DISC_BY_UUID_CHAR ● GATT_DISC_DESC_CHAR
in	*conhdl*	Connection handle.

Response:

GATT_DISC_CHAR_ALL_CMP_EVT or GATT_DISC_CHAR_BY_UUID_CMP_EVT or GATT_CMP_EVT

Description:

This API is used by the application to send a GATT_DISC_CHAR_REQ with the parameters deduced from the req_type. The definition for the different codes for req_type can be found in gatt.h. Upon reception of this message, GATT will checks whether the parameters are correct, if not correct then the GATT_CMP_EVT message will be generically built and sent to Application directly. An error status is also possible be either GATT_INVALID_PARAM_ERR or GATT_INVALID_TYPE_IN_SVC_SEARCH. If parameter is correct the GATT_DISC_CHAR_ALL_CMP_EVT or GATT_DISC_CHAR_BY_UUID_CMP_EVT message deceided by req_type will be received with the searched UUID, start handle and end handle together.

void app_gatt_disc_char_desc_req (uint16_t *conhdl*)

Parameters:

in	*conhdl*	Connection handle.

Response:

GATT_DISC_CHAR_DESC_CMP_EVT and GATT_CMP_EVT

Description:

This API is used by the application to send a GATT_DISC_CHAR_DESC_REQ mssage. Upon reception of this message, GATT will checks whether the parameters are correct, if not correct then the GATT_CMP_EVT message with error code GATT_INVALID_PARAM_ERR will be generically built and sent to Application directly. If parameter is correct, the GATT_DISC_CHAR_DESC_CMP_EVT message will be received.

void app_gatt_read_char_req (uint8_t *req_type*, uint16_t *conhdl*, uint16_t *valhdl*)

Parameters:

in	*req_type*	GATT request type: ● GATT_READ_CHAR ● GATT_READ_BY_UUID_CHAR ● GATT_READ_LONG_CHAR ● GATT_READ_MULT_LONG_CHAR ● GATT_READ_DESC ● GATT_READ_LONG_DESC
in	*conhdl*	Connection handle.
in	*valhdl*	Value handle.

Response:

GATT_READ_CHAR_RESP or GATT_READ_CHAR_MULTI_RESP

Description:

This API is used by the application to send a GATT_READ_CHAR_REQ mssage. Upon reception of this message, GATT will checks whether the parameters are correct, if not correct then the GATT_CMP_EVT message with error code GATT_INVALID_PARAM_ERR will be generically built and sent to Application directly. If parameter is correct, the GATT_READ_CHAR_RESP message will be received.

void app_gatt_write_char_req (uint8_t *req_type*, uint16_t *conhdl*, uint16_t *valhdl*, uint16_t *val_len*, uint8_t * *pdata*)

Parameters:

in	*req_type*	GATT request type: ● GATT_WRITE_NO_RESPONSE ● GATT_WRITE_SIGNED ● GATT_WRITE_CHAR ● GATT_WRITE_LONG_CHAR ● GATT_WRITE_DESC ● GATT_WRITE_LONG_DESC
in	*conhdl*	Connection handle.
in	*valhdl*	Value handle.
in	*val_len*	Value length.
in	*pdata*	Pointer to data.

Response:

GATT_WRITE_CHAR_RESP

Description:

This API is used by the application to send a GATT_WRITE_CHAR_REQ mssage. Upon reception of this message, GATT will checks whether the parameters are correct, if not correct then the GATT_CMP_EVT message with error code GATT_INVALID_PARAM_ERR will be generically built and sent to Application directly. If parameter is correct, the GATT_WRITE_CHAR_RESP message will be received.

void app_gatt_write_reliable_req (uint16_t *conhdl*, uint8_t *nb_writes*, uint8_t *auto_execute*, struct gatt_reliable_write * *data_write*)

Parameters:

in	*conhdl*	Connection handle.
in	*nb_writes*	Number of reliable writes.
in	*auto_execute*	Automatic execute write or not(0x00 don't execute, 0x01 write).
in	*data_write*	Pointer to the array of struct gatt_write_reliable_req.

Response:

GATT_WRITE_CHAR_RELIABLE_RESP or GATT_CMP_EVT

Description:

This API is used by the application to send a GATT_WRITE_RELIABLE_REQ mssage. Upon reception of this message, GATT will checks whether the parameters are correct, if not correct then the GATT_CMP_EVT message with error code GATT_INVALID_PARAM_ERR will be generically built and sent to Application directly. If parameter is correct, the GATT_WRITE_CHAR_RELIABLE_RESP message will be received.

void app_gatt_execute_write_char_req (uint16_t *conhdl*, uint8_t *exe_wr_ena*)

Parameters:

in	*conhdl*	Connection handle.
in	*exe_wr_ena*	Option flag to indicate for write or cancel(0x00 cancel, 0x01 write).

Response:

GATT_CANCEL_WRITE_CHAR_RESP

Description:

This API is used by the application to send a GATT_EXECUTE_WRITE_CHAR_REQ mssage. Upon reception of this message, GATT will checks whether the parameters are correct, if not correct then the GATT_CMP_EVT message with error code GATT_INVALID_PARAM_ERR will be generically built and sent to Application directly. If parameter is correct, the GATT_CANCEL_WRITE_CHAR_RESP message will be received if cancel the reliable write.

void app_gatt_notify_req (uint16_t *conhdl*, uint16_t *charhdl*)

Parameters:

in	*conhdl*	Connection handle.
in	*charhdl*	Characteristic value handle.

Response:

GATT_NOTIFY_CMP_EVT or none

Description:

This API is used by the application to send a GATT_NOTIFY_REQ mssage. Upon reception of this message, GATT will checks whether the parameters are correct, if not correct then the GATT_CMP_EVT message with error code GATT_INVALID_PARAM_ERR will be generically built and sent to Application directly. If permission is not allowed, the GATT_NOTIFY_CMP_EVT message will be received or application will not receive any mesage.

void app_gatt_indicate_req (uint16_t *conhdl*, uint16_t *charhdl*)

Parameters:

in	*conhdl*	Connection handle.
in	*charhdl*	Characteristic value handle.

Response:

GATT_HANDLE_VALUE_CFM

Description:

This API is used by the application to send a GATT_INDICATE_REQ mssage. Upon reception of this message, GATT will checks whether the parameters are correct, if not correct then the GATT_CMP_EVT message with error code GATT_INVALID_PARAM_ERR will be generically built and sent to Application directly. If parameter is correct, the GATT_HANDLE_VALUE_CFM message will be received.

4.2.2 Generic Attribute Profile Task API

Detailed Description

GATT Task APIs are used to handle the message from GATT or APP.

Data Structure Documentation

struct gatt_disc_svc_all_cmp_evt

Data Fields:

uint8_t	status	complete event status
uint8_t	nb_resp	number of value pairs
struct gatt_svc_list	list	contain data list

struct gatt_disc_svc_by_uuid_cmp_evt

Data Fields:

uint8_t	status	complete event status
uint8_t	nb_resp	number of value pairs
struct gatt_svc_range_list	list	list of found services

struct gatt_disc_cmp_evt

Data Fields:

uint8_t	status	complete event status

struct gatt_resource_access_rsp

Data Fields:

uint16_t	conhdl	device connection handle

Function Documentation

int app_gatt_disc_svc_all_cmp_evt_handler (ke_msg_id_t const *msgid*, struct gatt_disc_svc_all_cmp_evt const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GATT_DISC_SVC_ALL_CMP_EVT
in	*param*	Pointer to struct gatt_disc_svc_all_cmp_evt
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GATT

Returns:

If the message was consumed or not.

Description:

This handler is used by the Client role of GATT to receive the searched services from the remote server. The searched service items include the UUID, start handle and end handle.

Note:

GATT service list structure refer to struct gatt_svc_list

int app_gatt_disc_svc_by_uuid_cmp_evt_handler (ke_msg_id_t const *msgid*, struct gatt_disc_svc_by_uuid_cmp_evt const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GATT_DISC_SVC_BY_UUID_CMP_EVT
in	*param*	Pointer to struct gatt_disc_svc_by_uuid_cmp_evt
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GATT

Returns:

If the message was consumed or not.

Description:

This handler is used by the Client role of GATT to receive the searched services from the remote server. The searched service item include start handle and end handle.

Note:

GATT service list structure refer to struct gatt_svc_range_list

int app_gatt_disc_svc_incl_cmp_evt_handler (ke_msg_id_t const *msgid*, struct gatt_disc_svc_incl_cmp_evt const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GATT_DISC_SVC_INCL_CMP_EVT
in	*param*	Pointer to struct gatt_disc_svc_incl_cmp_evt
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GATT

Returns:

If the message was consumed or not.

Description:

This handler is used by the Client role of GATT to receive the searched services from the remote server. The searched service item include included service UUID, start handle and end handle.

int app_gatt_disc_svc_all_128_cmp_evt_handler (ke_msg_id_t const *msgid*, struct gatt_disc_svc_all_128_cmp_evt const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GATT_DISC_SVC_ALL_128_CMP_EVT
in	*param*	Pointer to struct gatt_disc_svc_all_128_cmp_evt
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GATT

Returns:

If the message was consumed or not.

Description:

This handler is used by the Client role of GATT to receive the searched services from the remote server. The searched service item include included service UUID, start handle and end handle.

int app_gatt_disc_char_all_cmp_evt_handler (ke_msg_id_t const *msgid*, struct gatt_disc_char_all_cmp_evt const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GATT_DISC_CHAR_ALL_CMP_EVT
in	*param*	Pointer to struct gatt_disc_char_all_cmp_evt
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GATT

Returns:

If the message was consumed or not.

Description:

This handler is used by the Client role of GATT to receive the searched characteristics from the remote server. The searched characteristics item include properties, pointer handle to UUID and characteristic UUID.

int app_gatt_disc_char_by_uuid_cmp_evt_handler (ke_msg_id_t const *msgid*, struct gatt_disc_char_by_uuid_cmp_evt const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GATT_DISC_CHAR_BY_UUID_CMP_EVT
in	*param*	Pointer to struct gatt_disc_char_by_uuid_cmp_evt
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GATT

Returns:

If the message was consumed or not.

Description:

This handler is used by the Client role of GATT to receive the searched characteristics from the remote server. The searched characteristics item include the UUID, properties, pointer handle to UUID and characteristic UUID.

int app_gatt_disc_char_all_128_cmp_evt_handler (ke_msg_id_t const *msgid*, struct gatt_disc_char_all_128_cmp_evt const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GATT_DISC_CHAR_ALL_128_CMP_EVT
in	*param*	Pointer to struct gatt_disc_char_all_128_cmp_evt
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GATT

Returns:

If the message was consumed or not.

Description:

This handler is used by the Client role of GATT to receive the searched characteristics from the remote server. The searched characteristics item include properties, pointer handle to UUID and characteristic UUID.

int app_gatt_disc_char_by_uuid_128_cmp_evt_handler (ke_msg_id_t const *msgid*, struct gatt_disc_char_by_uuid_128_cmp_evt const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GATT_DISC_CHAR_BY_UUID_128_CMP_EVT
in	*param*	Pointer to struct gatt_disc_char_by_uuid_128_cmp_evt
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GATT

Returns:

If the message was consumed or not.

Description:

This handler is used by the Client role of GATT to receive the searched characteristics from the remote server. The searched characteristics item include properties, pointer handle to UUID and characteristic UUID.

int app_gatt_disc_char_desc_cmp_evt_handler (ke_msg_id_t const *msgid*, struct gatt_disc_char_desc_cmp_evt const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GATT_DISC_CHAR_DESC_CMP_EVT
in	*param*	Pointer to struct gatt_disc_char_desc_cmp_evt
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GATT

Returns:

If the message was consumed or not.

Description:

This handler is used by the Client role of GATT to receive the searched characteristics descriptors from the remote server. The searched characteristics descriptors item include the database element handle and descriptor UUID.

int app_gatt_disc_char_desc_128_cmp_evt_handler (ke_msg_id_t const *msgid*, struct gatt_disc_char_desc_128_cmp_evt const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GATT_DISC_CHAR_ALL_128_CMP_EVT
in	*param*	Pointer to struct gatt_disc_char_desc_128_cmp_evt
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GATT

Returns:

If the message was consumed or not.

Description:

This handler is used by the Client role of GATT to receive the searched characteristics from the remote server. The searched characteristics item include properties, pointer handle to UUID and characteristic UUID.

int app_gatt_read_char_resp_handler (ke_msg_id_t const *msgid*, struct gatt_read_char_resp const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GATT_READ_CHAR_RESP
in	*param*	Pointer to struct gatt_read_char_resp
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GATT

Returns:

If the message was consumed or not.

Description:

This handler is used by the Client role of GATT to receive the value of the attribute handle element from the remote server. The element item include the data length and data.

int app_gatt_read_char_mult_resp_handler (ke_msg_id_t const *msgid*, struct gatt_read_char_mult_resp const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GATT_READ_CHAR_MULTI_RESP
in	*param*	Pointer to struct gatt_read_char_mult_resp
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GATT

Returns:

If the message was consumed or not.

Description:

This handler is used by the Client role of GATT to receive the multiple value of the attribute handle element from the remote server. The element item include the data length and data.

int app_gatt_write_char_resp_handler (ke_msg_id_t const *msgid*, struct gatt_write_char_resp const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GATT_WRITE_CHAR_RESP
in	*param*	Pointer to struct gatt_write_char_resp
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GATT

Returns:

If the message was consumed or not.

Description:

This handler is used by the Client role of GATT to handle the result of write characteristics value to the remote server.

int app_gatt_write_char_reliable_resp_handler (ke_msg_id_t const *msgid*, struct gatt_write_reliable_resp const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GATT_WRITE_CHAR_RELIABLE_RESP
in	*param*	Pointer to struct gatt_write_reliable_resp
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GATT

Returns:

If the message was consumed or not.

Description:

This handler is used by the Client role of GATT to handle the result of write long characteristics value to the remote server.

int app_gatt_cancel_write_char_resp_handler (ke_msg_id_t const *msgid*, void const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GATT_CANCEL_WRITE_CHAR_RESP
in	*param*	None
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GATT

Returns:

If the message was consumed or not.

Description:

This handler is used by the Client role of GATT to handle the result of cancel write characteristics request.

int app_gatt_notify_cmp_evt_handler (ke_msg_id_t const *msgid*, struct gatt_notify_cmp_evt const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GATT_NOTIFY_CMP_EVT
in	*param*	Pointer to struct gatt_notify_cmp_evt
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GATT

Returns:

If the message was consumed or not.

Description:

This handler is used by the Client role of GATT to indicate the failed reason of notify request.

int app_gatt_handle_value_notif_handler (ke_msg_id_t const *msgid*, struct gatt_handle_value_notif const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GATT_HANDLE_VALUE_NOTIF
in	*param*	Pointer to struct gatt_handle_value_notif
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GATT

Returns:

If the message was consumed or not.

Description:

This handler is used by the Client role of GATT to handle the value notification

int app_gatt_handle_value_ind_handler (ke_msg_id_t const *msgid*, struct gatt_handle_value_ind const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GATT_HANDLE_VALUE_IND
in	*param*	Pointer to struct gatt_handle_value_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GATT

Returns:

If the message was consumed or not.

Description:

This handler is used by the Client role of GATT to handle reception of a peer device indication.

int app_gatt_handle_value_cfm_handler (ke_msg_id_t const *msgid*, struct gatt_handle_value_cfm const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GATT_HANDLE_VALUE_CFM
in	*param*	Pointer to struct gatt_handle_value_cfm
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GATT

Returns:

If the message was consumed or not.

Description:

This handler is used by the Client role of GATT to Handles reception of a peer device confirmation of a previously sent indication.

int app_gatt_disc_cmp_evt_handler (ke_msg_id_t const *msgid*, struct gatt_disc_cmp_evt const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GATT_DISC_CMP_EVT
in	*param*	Pointer to struct gatt_disc_cmp_evt
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GATT

Returns:

If the message was consumed or not.

Description:

This handler is used by the GATT Discovery all services complete event.

int app_gatt_cmp_evt_handler (ke_msg_id_t const *msgid*, struct gatt_cmp_evt const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	GATT_CMP_EVT
in	*param*	Pointer to struct gatt_cmp_evt
in	*dest_id*	TASK_APP
in	*src_id*	TASK_GATT

Returns:

If the message was consumed or not.

Description:

This handler is used to tell application the complete of GATT operations.

4.3 Security Manager (SM)

4.3.1 Security Manager Protocol API

Detailed Description

The Security Manager allows two devices to setup a secure relationship either by encrypting a link, by bonding (exchanging information about each other) or signature use over a plain link. Please refer to the Bluetooth Core 4.0 specification for the SM requirements and protocol methods. Further details can be found in the BLE Host Software Functional Specification document [2]. Due to the possibility that a device be connected to several devices and that it starts the security procedure at different times simultaneously with some of them, the SM has been implemented in the format of a SM Manager task with one instance (SMPM) and a SM Controller task (SMPC) that will handle connection specific security procedures per connection. The SMPM environment will handle holding the keys that are unique per device and used for several connections, and act as a multiplexer between the SMPC tasks and the uniquely instantiated LLM for the encrypt functionality, or GAP for being informed of connections and disconnections.

SMP APIs are used by the Application directly especially for requesting keys necessary during the pairing/encrypting procedures.

Function Documentation

void app_smpm_set_key_req (uint8_t *key_code*, struct smp_key * *key*)

Parameters:

in	*key_code*	GATT request type for distinguishing between IRK and CSRK. ● SMP_KDIST_IDKEY ● SMP_KDIST_SIGNKEY
in	*key*	This structure contains a 16 octet array (U8[16]) in which the key value from the application is written MSB to LSB from index 0 to 15.

Response:

SMPM_SET_KEY_CFM

Description:

This API is used by the Application to set the device keys that are unique for the device and not connection dependent. The key_code parameter allows to use one single API message for setting either the IRK or the CSRK value ¨C 0x02 is used for IRK and 0x04 for CSRK because the definitions made in SMPC used in key distribution formatting are reused in SMPM. The value of the keys is sent from Application to SMPM in MSB to LSB format because these keys are used in the LE_Encrypt command in this format directly. Upon receiving a CSRK set request, the signature counter, also kept in the SMPM environment, will be reset to 0, and also the SMPM_GET_SUBKEYS_REQ is sent to SMPC to calculate the K1 and K2 keys needed in the signature generation algorithm, to save time when a signature will be asked from SM block. The SMPM_SET_KEY_CFM is sent to the API with a simple status and the key code value for confirming that the right key was set.

void app_smpc_tk_req_rsp (uint8_t *idx*, uint8_t *status*, struct smp_key * *tk*)

Parameters:

in	*idx*	Connection index for which the TK is sent from application.
in	*status*	OK if TK was found and input, not OK if key is not found by application.
in	*tk*	A 16 octet array (U8[16]) filled MSB to LSB from [0:15] because it is used with the LE_Encrypt command.

Response:

None

Description:

This message is used by the application to respond to SMPC_TK_REQ_IND message with either status OK and the TK value needed, or status not OK and all 0's in the TK parameter space. After receiving the SMCP_TK_REQ_IND, the application may have had more exchanges with even higher layers or the User in order to obtain the key, but that is up to the implementation.

void app_smpc_ltk_req_rsp (uint8_t *idx*, uint8_t *status*, uint8_t *sec_prop*, uint16_t *ediv*, struct rand_nb * *rand_nb*, struct smp_key * *ltk*)

Parameters:

in	*idx*	Connection index for which the LTK and its associated information is being given.
in	*status*	If OK, the information was retrieved by application, if not OK, the security procedure will stop.
in	*sec_prop*	Security properties of this LTK.
in	*ediv*	Encryption diversifier associated with the LTK.
in	*rand_nb*	Random number associated with the LTK, 8 octet array filled with the random number LSB to MSB [0:7].
in	*ltk*	16 octet array with the LTK value LSB to MSB [0:15].

Response:

None

Description:

This message is sent by the application in response to the SMPC_LTK_REQ_IND message. If the status is OK, then the rest of the parameters will be used for the procedure the SMPC task corresponding to the index is currently handling. If the status is not OK (key was not found), SMPC decides the following steps depending on the current procedure.

void app_smpc_irk_req_rsp (uint8_t *idx*, uint8_t *status*, struct bd_addr * *addr*, struct smp_key * *irk*)

Parameters:

in	*idx*	Connection index for which the IRK is being given.
in	*status*	If OK, the information was retrieved by application, if not OK, the security procedure will stop.
in	*addr*	6 octet array with the BD address associated with this IRK, LSB to MSB [0:5].
in	*irk*	16 octet array with the IRK value LSB to MSB [0:15].

Response:

None

Description:

This message is sent from application in response to SMPC_IRK_REQ_IND. If the status is OK, an IRK value was found and it is in the message parameters together with the BD address to which it was associated. SMPC will use it in trying to solve the random address it has under study, and if the IRK fails to match, a new one will be requested. If the status is not OK and no (more) IRKs exists in application, the SMPC will decide what the next step in the address solving procedure is.

void app_smpc_csrk_req_rsp (uint8_t *idx*, uint8_t *status*, uint8_t *sec_status*, struct smp_key * *csrk*)

Parameters:

in	*idx*	Connection index for which the CSRK is being given.
in	*status*	If OK, the information was retrieved by application, if not OK, the security procedure will stop.
in	*sec_status*	Security Status with connection index.
in	*csrk*	16 octet array with the CSRK value LSB to MSB [0:15].

Response:

None

Description:

This message is sent by application in response to SMPC_CSRK_REQ_IND and it contains the status of application search for a CSRK stored associated with the BD address indicated in the request, but also having had a last signature counter smaller than the one in the request. An appropriate status is sent in the response together with the found value of the CSRK or all 0's if the status is not OK.

void app_smpc_chk_bd_addr_req_rsp (uint8_t *idx*, uint8_t *found_flag*, uint8_t *sec_status*, uint8_t *type*, struct bd_addr * *addr*)

Parameters:

in	*idx*	Connection index for which the address check is received (authorization) may be for a free task.
in	*found_flag*	Found or not.
in	*sec_status*	Authentication, Authorization and Bonded status information recovered from application stored information about this BD address.

in	*type*	Type of address that was checked.
in	*addr*	6 octet array with the BD address that was checked.

Response:

None

Description:

This message is the response for SMPC_CHK_BD_ADDR_REQ_IND, informing SMPC that the bd address that was requested to be checked has been found or not, and if found, it gives the link the security property set in lk_sec_status parameter.

void app_smpc_start_enc_req (uint8_t *idx*, uint8_t *auth_req*, uint16_t *ediv*, struct rand_nb * *rand_nb*, struct smp_key * *ltk*)

Parameters:

in	*idx*	Connection index for which application want to start the encryption procedure with an existing LTK.
in	*auth_req*	Security properties of the LTK to be used..
in	*ediv*	Encryption diversifier associated with the LTK.
in	*rand_nb*	Random number associated with the LTK, 8 octet array filled with the random number LSB to MSB [0:7].
in	*ltk*	16 octet array with the LTK value LSB to MSB [0:15].

Response:

None

Description:

This message can be sent by the Higher Layers to directly encrypt a link with a peer using known bonding information from a previous connection when pairing+bonding occurred. The known information for direct encryption is the LTK and its associated EDIV and Random Number values. The status of the encryption procedure is returned to the application using the SMPC_ENC_STARTED_IND.

4.3.2 Security Manager Protocol Task API

Detailed Description

SMP Task APIs are used to handle the message from SMPM, SMPC or APP.

Data Structure Documentation

struct smpc_sec_started_ind

Data Fields:

uint8_t	idx	Connection index.
uint8_t	status	Status (OK or failure status)
uint8_t	key_size	Key size for the LTK/STK agreed upon during pairing features exchange)
uint8_t	sec_prop	Security properties of the keys.
uint8_t	bonded	Bonding status.

struct smpc_tk_req_ind

Data Fields:

uint8_t	idx	Connection index.
uint8_t	oob_en	key type: OOB 16B or 6 digit
uint8_t	disp_en	action expected if 6 digit, depending on IOs

struct smpc_ltk_req_ind

Data Fields:

uint8_t	idx	Connection index.
uint8_t	auth_req	Authentication Requirements from request.

struct smpc_irk_req_ind

Data Fields:

uint8_t	idx	Connection index.

struct smpc_csrk_req_ind

Data Fields:

uint8_t	idx	Connection index.
struct bd_addr	addr	Bd address of device for which bonding info containing CSRK should exist.
uint32_t	signcnt	Signing counter received - to check against last known in APP.

struct smpc_key_ind

Data Fields:

uint8_t	idx	Connection index.
uint8_t	key_code	Key code - use one of defined values for Key distribution parameters.
uint16_t	ediv	EDIV (=0 if not sending an LTK)
struct rand_nb	nb	Random number (=0 if not sending an LTK)
struct smp_key	key	Key being sent to Host (LTK/IRK/CSRK)

struct smpc_chk_bd_addr_req

Data Fields:

uint8_t	idx	Connection index -may be a free task index.
uint8_t	type	Type of address to check.
struct bd_addr	addr	Random address to resolve or Public address to check in APP.

struct smpc_timeout_evt

Data Fields:

uint8_t	idx	

struct smpm_set_key_cfm

Data Fields:

uint8_t	status	Key set status.
uint8_t	key_code	Key code (irk or csrk - use those from Key distribution)

Function Documentation

int app_smpm_set_key_cfm_handler (ke_msg_id_t const *msgid*, struct smpm_set_key_cfm const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	SMPM_SET_KEY_CFM
in	*param*	Pointer to struct smpm_set_key_cfm
in	*dest_id*	TASK_APP
in	*src_id*	TASK_SMPM

Returns:

If the message was consumed or not.

Description:

This API is used by the SMPM to respond to the application to its SMPM_SET_KEY_REQ, informing it that saving the key values was done and the other actions related to setting a new key were initiated.

Note:

key_code:

- SMP_KDIST_IDKEY ///IRK (ID key)in distribution
- SMP_KDIST_SIGNKEY ///CSRK(Signature key) in distribution

int app_smpc_sec_started_ind_handler (ke_msg_id_t const *msgid*, struct smpc_sec_started_ind const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	SMPC_SEC_STARTED_IND
in	*param*	Pointer to struct smpc_sec_started_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_SMPC

Returns:

If the message was consumed or not.

Description:

This API is used to inform the application that the status of a security procedure.

int app_smpc_key_ind_handler (ke_msg_id_t const *msgid*, struct smpc_key_ind const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	SMPC_KEY_IND
in	*param*	Pointer to struct smpc_key_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_SMPC

Returns:

If the message was consumed or not.

Description:

This message is sent by SMPC during TKDP to application with the value of received bonding information from peer device: either LTK+EDIV+random number, or IRK, or CSRK. These values should be retrievable by the application at a later time if the implementation chosen allows it. The key_code can have one of the defined values for the presence of a key in a key distribution. The ediv and nb will be filled with 0's if it is an IRK or CSRK that is being sent to application.

Note:

Random number structure refer to struct rand_nb SMP key structure refer to struct smp_key

int app_smpc_tk_req_ind_handler (ke_msg_id_t const *msgid*, struct smpc_tk_req_ind const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	SMPC_TK_REQ_IND
in	*param*	Pointer to struct smpc_tk_req_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_SMPC

Returns:

If the message was consumed or not.

Description:

This message is sent by SMPC to application during pairing, when the TK is necessary for calculations of the security values involved in the procedure. Two flags indicate to application whether the necessary TK value is an OOB 16 octet value that the device should have, or if it is a simple PIN key, whether it should be input/displayed by the user/device. The values of the flags are determined after the pairing features exchange stage in the procedure. The status in the response from application will reflect whether the requested key was found, and will allow the procedure to continue or stop indicating the failure reason to the peer.

int app_smpc_ltk_req_ind_handler (ke_msg_id_t const *msgid*, struct smpc_ltk_req_ind const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	SMPC_LTK_REQ_IND
in	*param*	Pointer to struct smpc_ltk_req_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_SMPC

Returns:

If the message was consumed or not.

Description:

This message may be sent by SMPC to application on various occasions:

When the SMPC in Slave role receives a LE_LTK_Request event from LLC during encryption procedure, in which case the needed LTK must be retrieved from application.

When a Master device receives a Security Request PDU from the slave, to check whether an LTK with the right security properties exists is stored in application in order to encrypt the link and not pair ¨C this is when the Authentication Requirements parameters received in the Security Request from the Slave will allow the Application to decide whether it has encryption information corresponding to the requested level of security requested by Slave.

During TKDP when LTK needs to be distributed, it must be retrieved from application. In case the key is not retrieved, the response message will have a status reflecting that and SMPC will decide of the next step in the procedure. If it is, the key will be used either for encryption or distribution.

int app_smpc_irk_req_ind_handler (ke_msg_id_t const *msgid*, struct smpc_irk_req_ind const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	SMPC_IRK_REQ_IND
in	*param*	Pointer to struct smpc_irk_req_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_SMPC

Returns:

If the message was consumed or not.

Description:

This message is sent by SMPC during the procedure for solving a peer random address. The application are asked to deliver an IRK as long as they no longer hold any record of IRKs for the known devices that have bonded with the local device, or until the address has been solved.

int app_smpc_csrk_req_ind_handler (ke_msg_id_t const *msgid*, struct smpc_csrk_req_ind const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	SMPC_CSRK_REQ_IND
in	*param*	Pointer to struct smpc_csrk_req_ind
in	*dest_id*	TASK_APP
in	*src_id*	TASK_SMPC

Returns:

If the message was consumed or not.

Description:

This message is sent by SMPC when it needs the CSRK value associated to a device that just sent a signed message, in order to verify the signature. SMPC will extract the sign counter from the received message and also send the request including the BD address of the device that is sending signed messages. The SMPC_CSRK_REQ_RSP is received with a status code reflecting whether a CSRK value was found stored for that device or not, but also whether the Signature counter value is valid (larger than the last stored one for an existing CSRK).

int app_smpc_chk_bd_addr_req_ind_handler (ke_msg_id_t const *msgid*, struct smpc_chk_bd_addr_req const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	SMPC_CHK_BD_ADDR_REQ_IND
in	*param*	Pointer to struct smpc_chk_bd_addr_req
in	*dest_id*	TASK_APP
in	*src_id*	TASK_SMPC

Returns:

If the message was consumed or not.

Description:

This message is sent by SMPC to application when a SMPC_SOLVE_ADDR_REQ is received for a peer address. Application is supposed to check the BD address existence in stored records and tell SMPC what properties it has using the SMPC_CHK_BD_ADDR_REQ_RSP. Or resolve it if it is a random resolvable address by searching which stored IRK will match.

int app_smpc_timeout_evt_handler (ke_msg_id_t const *msgid*, struct smpc_timeout_evt const * *param*, ke_task_id_t const *dest_id*, ke_task_id_t const *src_id*)

Parameters:

in	*msgid*	SMPC_TIMEOUT_EVT
in	*param*	Pointer to struct smpc_timeout_evt
in	*dest_id*	TASK_APP
in	*src_id*	TASK_SMPC

Returns:

If the message was consumed or not.

Description:

This message is sent by SMPC when Timeout happens in SM procedures if more than 30s elapse between sending two commands. All SM PDUs are commands, so every time one is sent, a timer starts and will be reset when the next one is sent. It will be stopped of course when the procedure completes correctly. This timer is normally sufficient for the PDU exchange, the only vulnerable moment is PassKey entry if any is required and user is not fast enough. After a timeout, the SM L2CAP channel must not be reused until link re-establishment, so the SMPC task will become FREE and ignore all other SM local or peer requests, it's up to the application to disconnect when it has finished other actions.

Release History

REVISION	CHANGE DESCRIPTION	DATE
0.1	Initial release	2012-12-17
0.2	Update on Driver API descriptions	2013-4-16
0.4	Update on API for SDK v0.9	2013-7-12
0.7	Update on Driver API descriptions for SDK v0.9.6	2013-11-27
0.8	Update on Driver API descriptions for SDK v0.9.8	2014-01-10
0.9	Update on Driver API descriptions for SDK v1.2.0	2014-04-10
1.0	Modify ADC Driver API	2014-05-20
1.1	Modify ADC and I2C Driver API	2015-01-23