RT58x SDK

IOT Application Overview

V1.0

About this Document

This document supports “RT58x_SDK_v0.3.1” and later version.

Table of Contents

1. Introduction

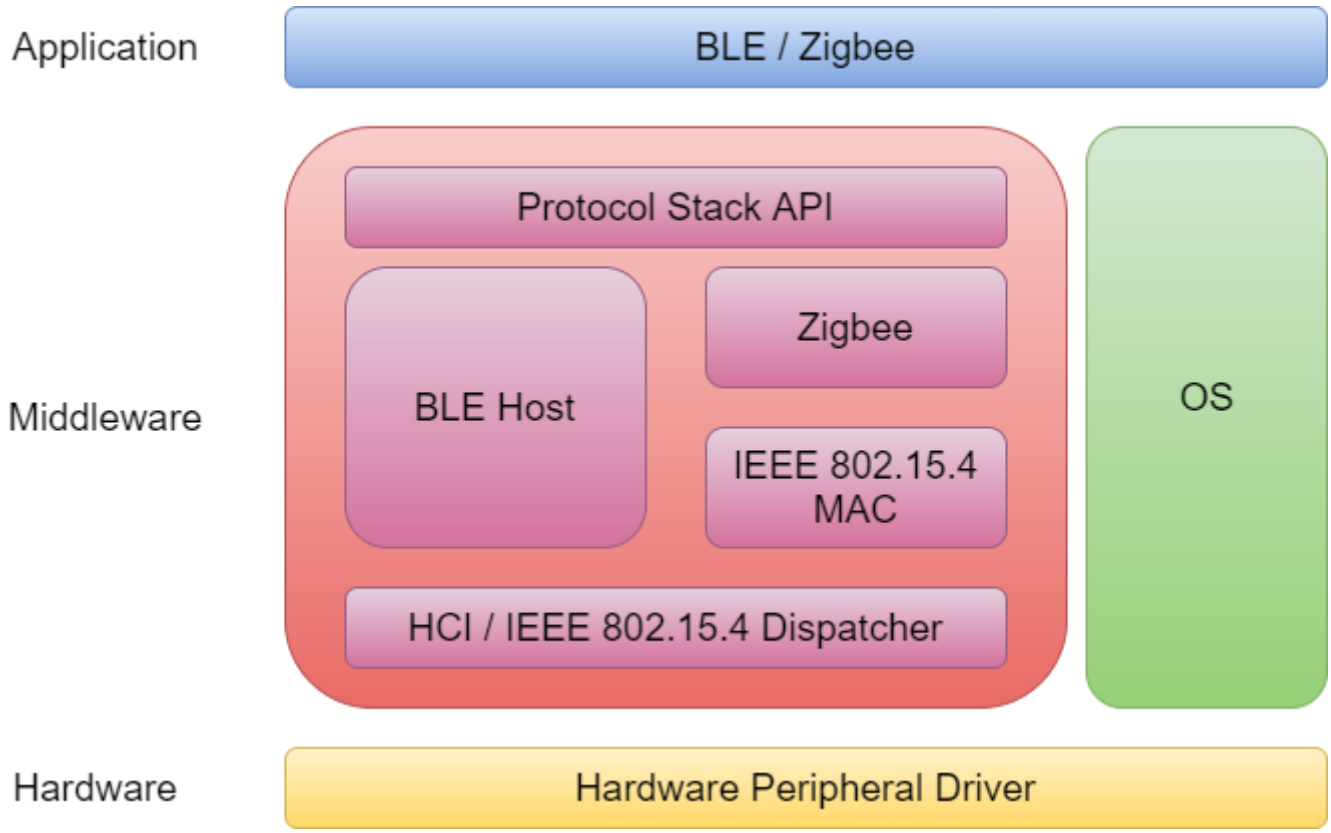
The RT58x SDK provides a rich IOT communication protocol libraries and application demos for user to build their products easily and quickly in time to market. In protocols, this SDK provides Zigbee, BLE, and an amazing dual mode with Zigbee and BLE.

**Please be informed that the bootloader must be installed in EVK first before installing and running the built applications.

2. Application

All RT58x applications are running on a real time OS with multiple tasks. Users could use the application projects included in SDK or refer them to build their own application to test the platform. The application could run in single task or more tasks. It depends the application necessary.

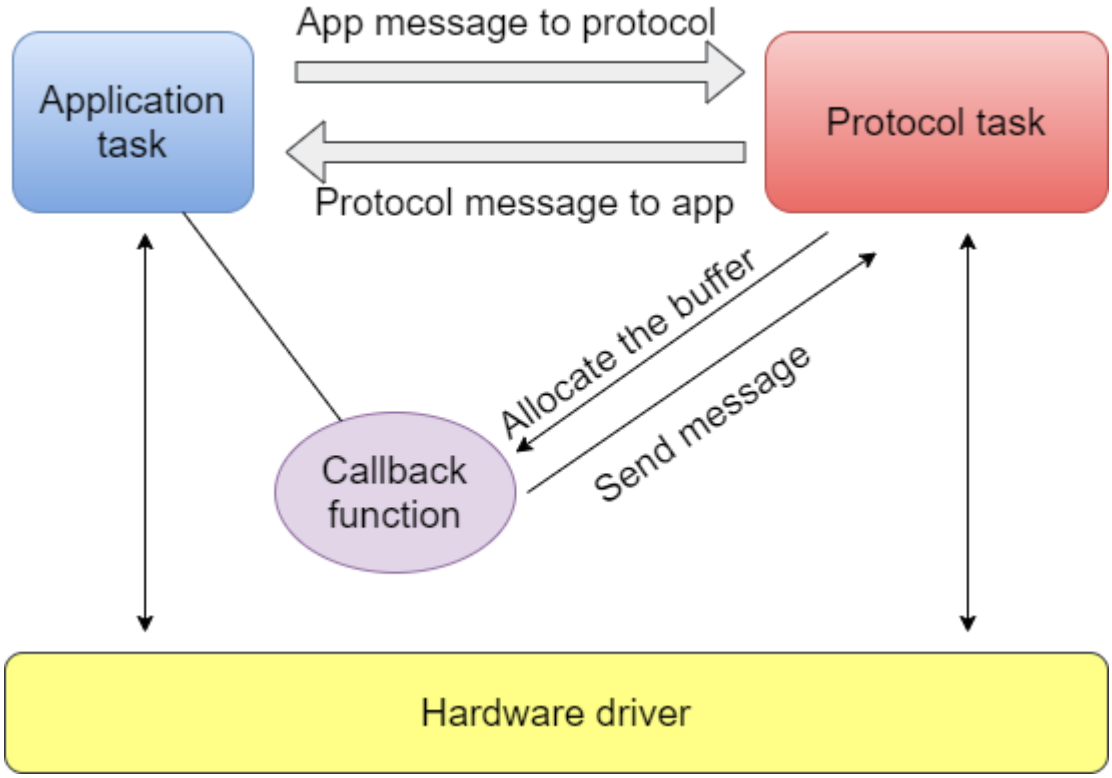
2.1 Application architecture



System running the application contains four parts, application, middleware, OS, and hardware driver. The stack view is shown on the following figure. The application is the top level and created by user running the necessary functions of product. The middleware is the BLE/Zigbee protocol libraries to handle the BLE/Zigbee communication event. The OS is to help the tasks

scheduling and management. The hardware driver is the function collections to help user to access hardware peripheral devices.

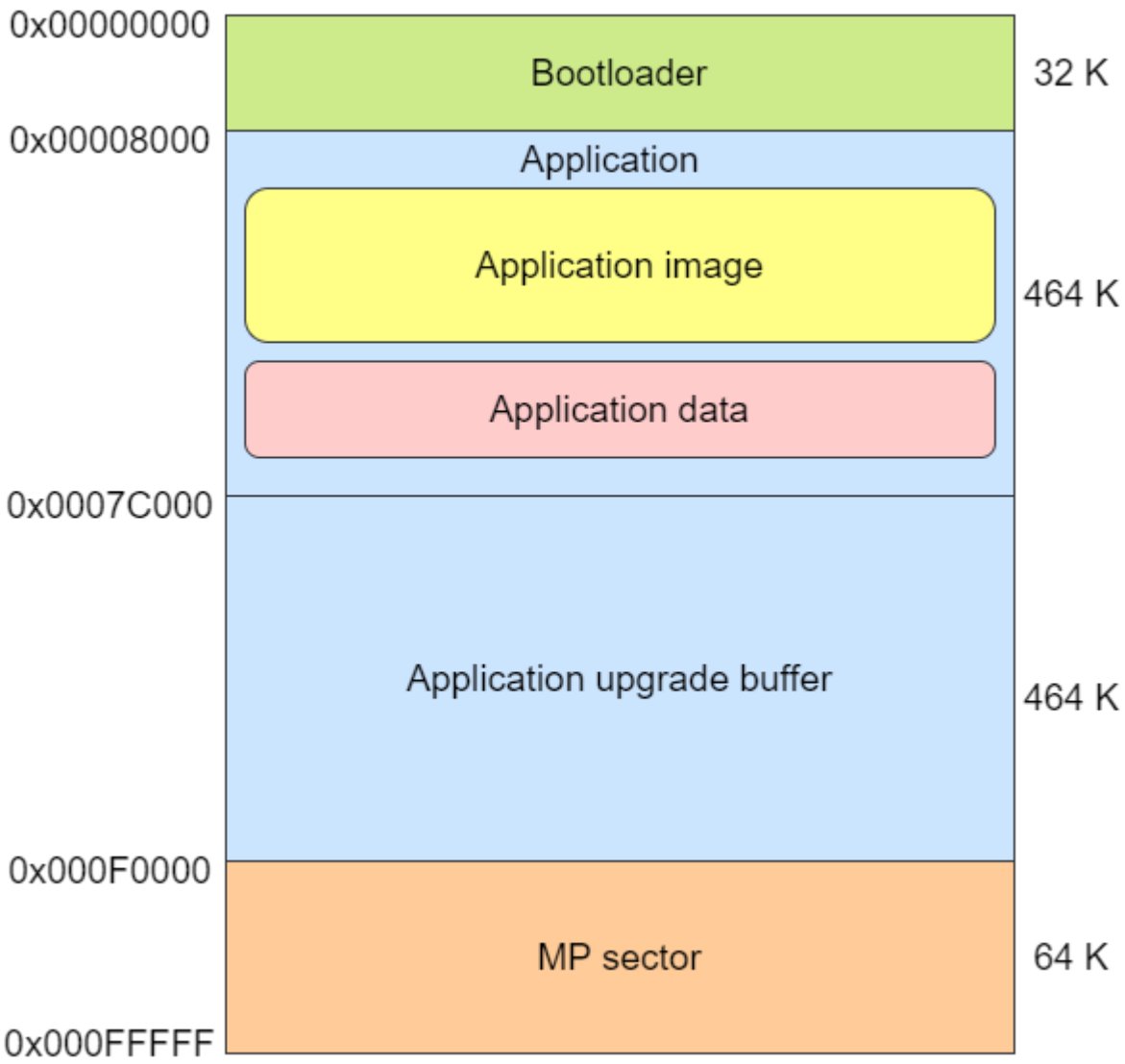
2.2 Application operation



The application and protocol stack are in different tasks. The communication between these tasks are through the message queue. When users create the application, they also need to create the queen and allocate the buffer for message exchange. In order to follow the “who allocate buffer, who release buffer” management policy, the application must provide a callback function for protocol stack to call for allocating the buffer and sending message.

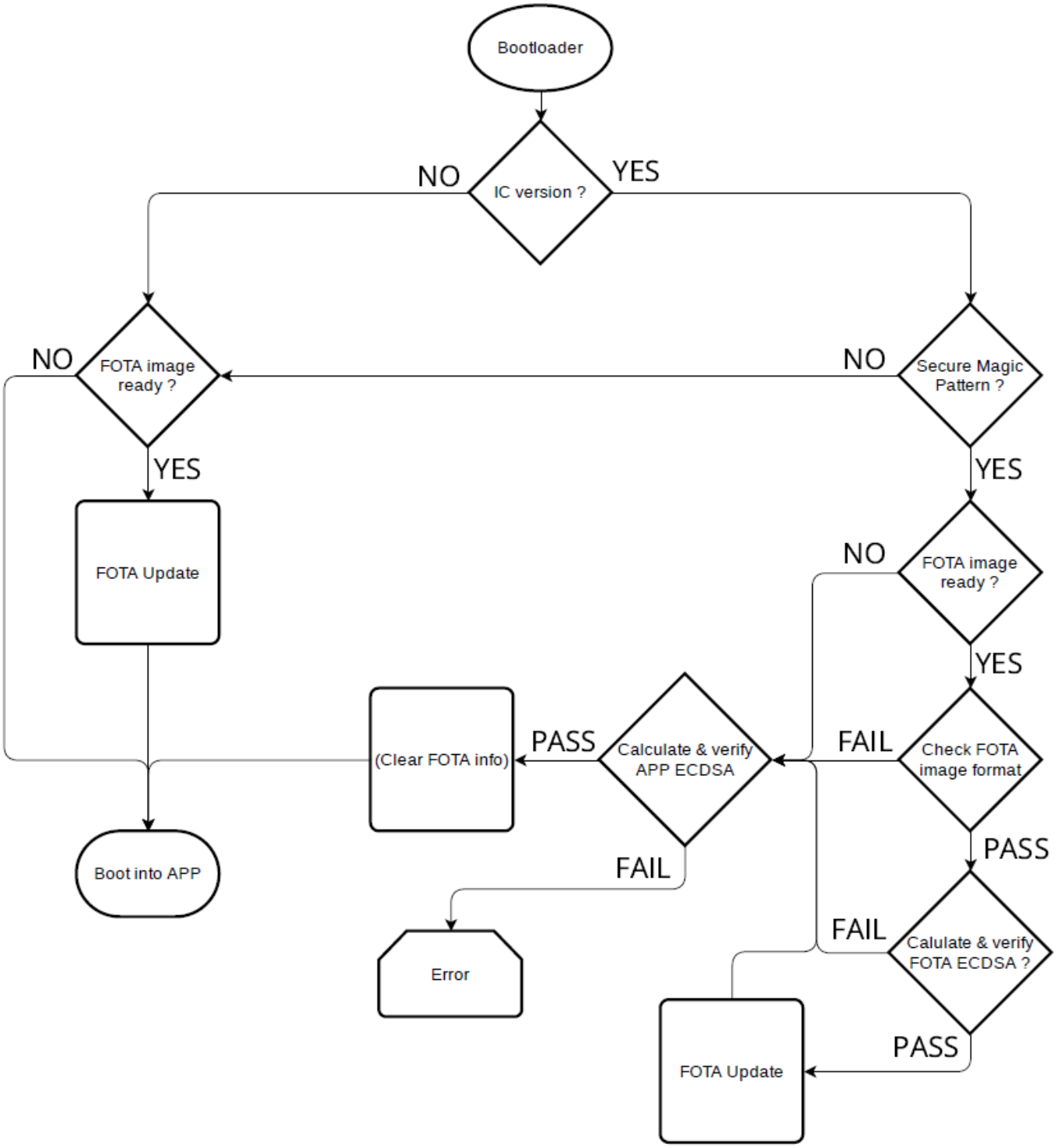
3. System flash map

RT58x's 1M flash is partitioned into 4 regions. They are bootloader, application, application upgrade buffer, and MP sector.



3.1 Bootloader

The bootloader is responsible for new upgraded image check and application image overwrite. The bootloader size is fixed and kept at 32K bytes. The image upgrade flow is shown as following figure.



3.2 Application

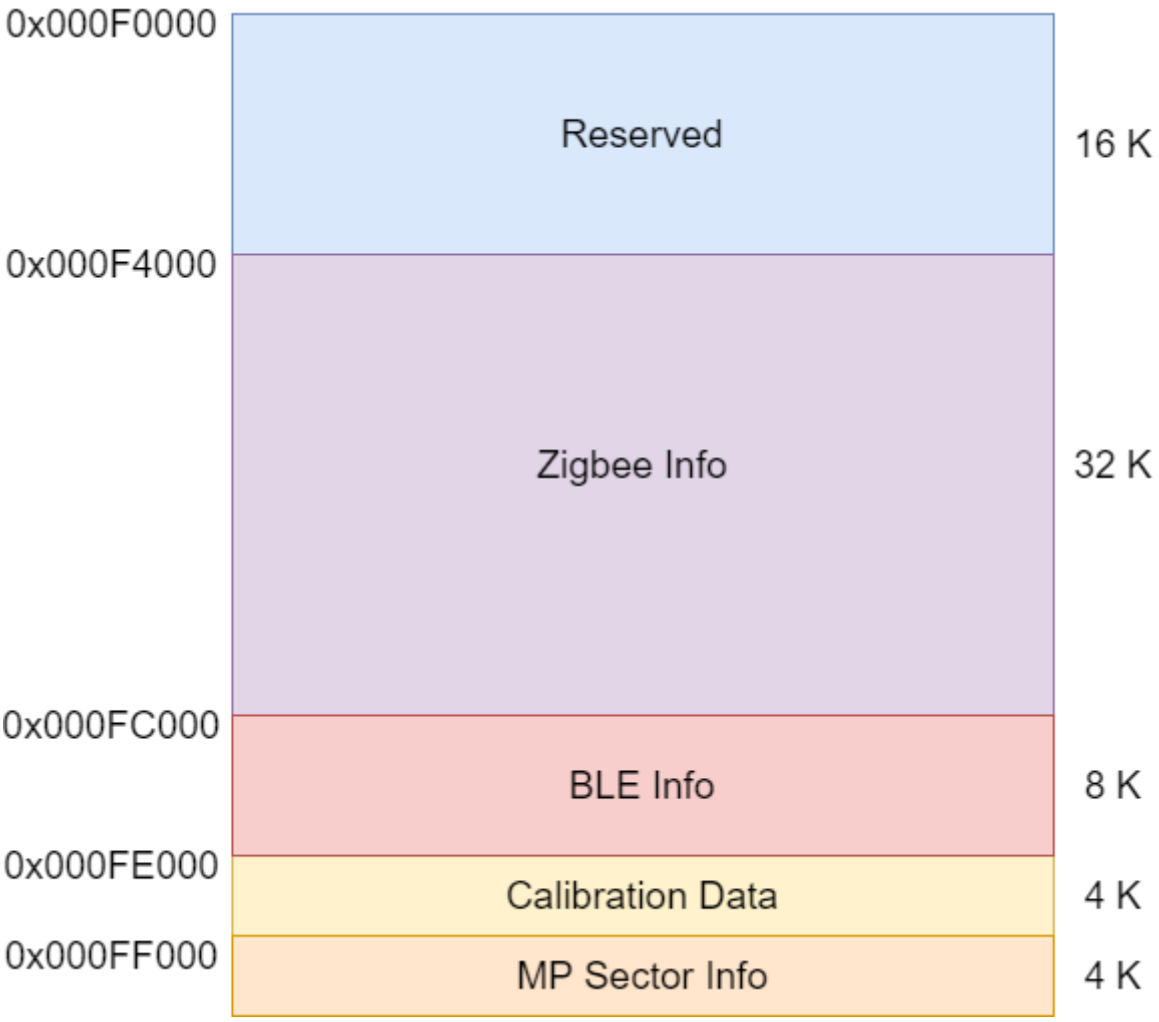
The application region is location for executing application image and data used by application. The total size for application and application data must keep less than 432K bytes. The application entry point address must at 0x8000.

3.3 Application upgrade buffer

The application upgrade buffer is used for storing the upgrade image when a new version image has found by system and downloaded.

3.4 MP sector

MP Sector has the structure in 64K bytes as the following figure. It has MP sector information in 4K bytes, calibration data in 4K bytes, the BLE stack running data in 8K bytes and the Zigbee running data in 32K bytes. The 16K bytes is reserved for the future use. It strongly recommends user to execute the MP calibration procedure before executing the application. It will make sure system running more reliably. The MP calibration will include crystal, RF, and power calibrations. The detail operation guideline could be found in the document:
"RT58x_MPTCB_and_MP_Tool_User_Guide".



4. OS function API

RT58x also provides a lot of its own OS API function for user to create and maintain the application task. All related functions could be found in file “sys_arch.c” under SDK directory “Middleware/Portable/System”.

The following sections are some examples for the reference. More functions and descriptions could refer the file “sys_arch.c”.

4.1 Task function

4.1.1 sys_task_new

```
/** creat task
 * @param name A descriptive name for the thread
 * @param thread Pointer to the thread entry function.
 * @param arg Pointer that will be used as the parameter for the thread being created.
 * @param u32_stacksize The size of the thread stack specified as the number of
 *        variables the stack can hold - not the number of bytes.
 * @param u32_priority The priority at which the thread should run.
 *        It must less than CONFIG_FREERTOS_MAX_PRIORITIES
 */
sys_task_t sys_task_new(const char *name,
                        sys_thread_fn thread,
                        void *arg,
                        uint32_t u32_stacksize,
                        uint32_t u32_priority
                        )
```

4.1.2 sys_thread_priority_get

```
/** get the thread priority
 * @param thread_handle
 * @return thread priority
 */
uint32_t sys_thread_priority_get(sys_task_t thread_handle)
```

4.1.3 sys_thread_priority_set

```
/** set the thread priority
 * @param thread_handle
```

```
 * @param u32_priority The priority at which the thread should run.
 *         It must less than CONFIG_FREERTOS_MAX_PRIORITIES
 */
void sys_thread_priority_set(sys_task_t thread_handle, uint32_t u32_priority)
```

4.2 Queue function

4.2.1 sys_queue_new

```
/** Creates an empty queue.
 * @param queue The queue handle
 * @param queue_length the length of queue
 * @param item_size the size of item of queue
 */
err_t sys_queue_new(sys_queue_t *queue, uint32_t queue_length, uint32_t item_size)
```

4.2.2 sys_queue_free

```
/** Deallocates a queue. If there are messages still present in the
 *   queue when the queue is deallocated, it is an indication of a
 *   programming error and the developer should be notified.
 * @param queue The queue handle
 */
void sys_queue_free(sys_queue_t *queue)
```

4.2.3 sys_queue_send

```
/** Send the "msg" to the queue.
 * @param queue The queue handle
 * @param msg The pointer of the msg
 */
void sys_queue_send(sys_queue_t *queue, void *msg)
```

4.2.4 sys_queue_send_from_isr

```
void sys_queue_send_from_isr(sys_queue_t *queue, void *msg)
```

4.2.5 sys_queue_send_with_timeout

```
/** Send the "msg" to the queue.
 * @param queue The queue handle
```

```
 * @param msg The pointer of the msg
 * @param u32_timeout millisecond timer
 */
err_t sys_queue_send_with_timeout(sys_queue_t *queue, void *msg, uint32_t u32_timeout)
```

4.2.6 sys_queue_recv

```
/** Blocks the thread until a message arrives in the queue, but does
 *   not block the thread longer than "timeout" milliseconds. The "msg" argument is a result
 *   parameter that is set by the function (i.e., by doing "*msg =
 *   ptr"). The "msg" parameter maybe NULL to indicate that the message
 *   should be dropped.
 * @param queue The queue handle
 * @param msg The pointer of the msg
 * @return The return values are the same as for the sys_arch_sem_wait()
 *             function: Number of milliseconds spent waiting or
 *             SYS_ARCH_TIMEOUT if there was a timeout.
 */
uint32_t sys_queue_recv(sys_queue_t *queue, void *msg, uint32_t u32_timeout)
```

4.2.7 sys_queue_remaining_size

```
/** Get the number of free spaces in a queue.
 * @param queue The queue handle
 */
uint32_t sys_queue_remaining_size(sys_queue_t *queue)
```

4.3 Memory function

4.3.1 sys_malloc_fn

```
/** Memory allocation
 * @param u32_size sleep time in milliseconds
 */
void *sys_malloc_fn(uint32_t u32_size, const char *pc_func_ptr, uint32_t u32_line)
```

4.3.2 sys_free_fn

```
/** Memory free
```

```
 * @param p_pointer
 */
void sys_free_fn(void *p_pointer, const char *pc_func_ptr, uint32_t u32_line)
```

Revision History

Revision	Description	Owner	Date
1.0	Initial version	Joshua	2022/02/11