BG95&BG77-QuecOpen RAM Memory Management

LPWA Module Series

Rev. BG95&BG77-QuecOpen_RAM_Memory_Management_V1.0

Date: 2019-08-29

Status: Preliminary

www.quectel.com

Our aim is to provide customers with timely and comprehensive service. For any assistance, please contact our company headquarters:

Quectel Wireless Solutions Co., Ltd.
Building 5, Shanghai Business Park Phase III (Area B), No.1016 Tianlin Road, Minhang District, Shanghai, China 200233
Tel: +86 21 5108 6236
Email: info@quectel.com

Or our local office. For more information, please visit:
http://www.quectel.com/support/sales.htm

For technical support, or to report documentation errors, please visit:
http://www.quectel.com/support/technical.htm
Or email to: support@quectel.com

About the Document

History

Revision	Date	Author	Description
1.0	2019-08-29	Justice HAN	Initial

Contents

1 Introduction

This document mainly introduces the memory management mechanism of user space RAM. Quectel QuecOpen solution is based on ThreadX^{TM}, so customers can use the ThreadX API directly to manage RAM memory in user space.

ThreadX memory byte pools are similar to a standard C heap. Unlike the standard C heap, it is possible to have multiple memory byte pools. Each memory byte pool is a public resource. ThreadX^{TM} places no constraints on how pools are used, except that memory byte services cannot be called from ISRs. In addition, threads will suspend on a pool until the requested memory is available.

This document mainly applies to Quectel BG95 and BG77 module series.

2 ThreadX APIs for Memory Byte Pool

These following APIs are used to manage ThreadX memory byte pool for QuecOpen application. Please refer to ***Chapter 3.1.3***, ***Chapter 3.2.3***, ***Chapter 3.3.2***, ***Chapter 3.5.2 and Chapter 3.6.1*** for details.

● *tx_byte_pool_create*	for creating a memory byte pool.
● *tx_byte_allocate*	for allocating memory from the specified memory byte pool.
● *tx_byte_pool_info_get*	for retrieving information about the specified memory byte pool.
● *tx_byte_release*	for releasing a previously allocated memory back to its associated pool.
● *tx_byte_pool_delete*	for deleting the specified memory byte pool.

3 Services of Memory Byte Pools

3.1. Create Memory Byte Pools

3.1.1. Overview

Memory byte pools are created either during initialization or during run-time by application threads. There is no limitation on the number of memory byte pools in an application.

3.1.2. Memory Areas of the Pool

The memory area for a memory byte pool is specified during creation. Like other memory areas in ThreadX^{TM}, it can be located anywhere in the target's address space. This is an important feature because of the considerable flexibility given to the application. For example, if the target hardware has a high-speed memory area and a low-speed memory area, the user can manage memory allocation for both areas by creating a pool in each of them.

3.1.3. ThreadX API tx_byte_pool_create

- **Prototype**

```
UINT tx_byte_pool_create(TX_BYTE_POOL *pool_ptr, CHAR *name_ptr, VOID *pool_start, ULONG
pool_size)
```

- **Description**

This service creates a memory pool in the area specified. Initially the pool consists of basically one very large free block. However, the pool breaks into smaller blocks as allocations are made.

- **Input Parameters**

pool_ptr:
Pointer to a memory pool control block.

name_ptr:
Pointer to the name of the memory pool.

pool_start:
Starting address of the memory pool.

pool_size:
Total number of bytes available for the memory pool.

- **Return Values**

TX_SUCCESS	(0x00)	Successful memory pool creation.
TX_POOL_ERROR	(0x02)	Invalid memory pool pointer. Either the pointer is NULL or the pool is already created.
TX_PTR_ERROR	(0x03)	Invalid starting address of the pool.
TX_SIZE_ERROR	(0x05)	Size of pool is invalid.
TX_CALLER_ERROR	(0x13)	Invalid caller of this service.

- **Example**

```
TX_BYTE_POOL my_pool;
UINT status;

/* Create a memory pool whose total size is 2000 bytes starting at address 0x500000. */
status = tx_byte_pool_create(&my_pool, "my_pool_name", (VOID *) 0x500000, 2000);
/* If status equals TX_SUCCESS, my_pool is available for allocating memory. */
```

3.2. Allocate Memory from Memory Byte Pools

3.2.1. Overview

Allocations from memory byte pools are similar to traditional malloc calls, which include the amount of memory desired (in bytes). Memory is allocated from the pool in a first-fit manner, i.e., the first free memory block that satisfies the request is used. Excess memory from this block is converted into a new block and placed back in the free memory list. This process is called fragmentation. Adjacent free memory blocks are merged together during a subsequent allocation search for a large enough free memory block. This process is called de-fragmentation.

3.2.2. Pool Capacity

The number of allocatable bytes in a memory byte pool is slightly less than what has been specified during creation. This is because management of the free memory area introduces some overhead. Each free memory block in the pool requires the equivalent of two C pointers of overhead. For example, if customers allocate 1000 bytes from memory byte pool, the available bytes of the memory byte pool will be reduced by 1000+8 bytes.

In addition, the pool is created with two blocks, a large free block and a small permanently allocated block at the end of the memory area. This allocated block is used to improve performance of the allocation algorithm. It eliminates the need to continuously check for the end of the pool area during merging.

During run-time, the amount of overhead in the pool typically increases. Allocations of an odd number of bytes are padded to insure proper alignment of the next memory block. In addition, overhead increases as the pool becomes more fragmented.

3.2.3. ThreadX API tx_byte_allocate

- **Prototype**

```
UINT tx_byte_allocate(TX_BYTE_POOL *pool_ptr, VOID **memory_ptr, ULONG memory_size,
ULONGwait_option)
```

- **Description**

This service allocates the specified number of bytes from the specified byte memory pool.

- **Input Parameters**

pool_ptr:
Pointer to a previously created memory pool.

memory_ptr:
Pointer to a destination memory pointer. On successful allocation, the address of the allocated memory area is placed where this parameter points to.

memory_size:
Number of bytes requested.

wait_option:
Defines how the service behaves if there is not enough memory available. The wait options are defined as follows:

TX_NO_WAIT	(0x00000000) Results in an immediate return from this service regardless of whether or not it was successful. This is the only valid option if the service is called from initialization.
TX_WAIT_FOREVER	(0xFFFFFFFF) Causes the calling thread to suspend indefinitely until enough memory is available.
timeout value	(0x00000001 through 0xFFFFFFFE) Specifies the maximum number of timer-ticks to stay suspended while waiting for the memory.

- **Return Values**

TX_SUCCESS	(0x00)	Successful memory allocation.
TX_DELETED	(0x01)	Memory pool was deleted while thread was suspended.

TX_NO_MEMORY	(0x10)	Service was unable to allocate the memory.
TX_WAIT_ABORTED	(0x1A)	Suspension was aborted by another thread, timer, or ISR.
TX_POOL_ERROR	(0x02)	Invalid memory pool pointer.
TX_PTR_ERROR	(0x03)	Invalid pointer to destination pointer.
TX_WAIT_ERROR	(0x04)	A wait option other than *TX_NO_WAIT* was specified on a call from a nonthread.
TX_CALLER_ERROR	(0x13)	Invalid caller of this service.

- **Example**

```
TX_BYTE_POOL my_pool;
unsigned char *memory_ptr;
UINT status;

/* Allocate a 112 byte memory area from my_pool. Assume that the pool has already been created with a
call to tx_byte_pool_create. */
status = tx_byte_allocate(&my_pool, (VOID **)&memory_ptr, 112, TX_NO_WAIT);
/* If status equals TX_SUCCESS, memory_ptr contains the address of the allocated memory area.
```

3.2.4. Un-deterministic Behavior

Although memory byte pools provide the most flexible memory allocation, they also suffer from somewhat un-deterministic behavior. For example, a memory byte pool may have 2,000 bytes of memory available but may not be able to satisfy an allocation request of 1,000 bytes. This is because there are no guarantees on how many of the free bytes are contiguous. Even if a 1,000 bytes free block exits, there are no guarantees on how long it may take to find the block. It is completely possible that the entire memory pool would need to be searched in order to find the 1,000 bytes block.

Because of this, it is generally good practice to avoid using memory byte services in areas where deterministic, real-time behavior is required. Many applications pre-allocate their required memory during initialization or run-time configuration.

3.3. Characteristics of Memory Byte Pools

3.3.1. Overview

The characteristics of each memory byte pool can be found in its control block. It contains useful information such as the number of available bytes in the pool. This structure is defined in the *tx_api.h* file.

However, customers can not directly access this structure in user space, since the address of this structure is a kernel address. API *tx_byte_pool_info_get* is used to get characteristics of each memory byte pools, refer to the next paragraph.

3.3.2. ThreadX API tx_byte_pool_info_get

- **Prototype**

```
UINT tx_byte_pool_info_get(TX_BYTE_POOL *pool_ptr, CHAR **name, ULONG *available, ULONG
*fragments, TX_THREAD **first_suspended, ULONG *suspended_count, TX_BYTE_POOL **next_pool)
```

- **Description**

This service retrieves information about the specified memory byte pool.

- **Input Parameters**

pool_ptr:
Pointer to previously created memory pool.

name:
Pointer to destination for the pointer to the byte pool's name.

available:
Pointer to destination for the number of available bytes in the pool.

fragments:
Pointer to destination for the total number of memory fragments in the byte pool.

first_suspended:
Pointer to destination for the pointer to the thread that is first on the suspension list of this byte pool.

suspended_count:
Pointer to destination for the number of threads currently suspended on this byte pool.

next_pool:
Pointer to destination for the pointer of the next created byte pool.

- **Return Values**

TX_SUCCESS	(0x00)	Successful pool information retrieve.
TX_POOL_ERROR	(0x02)	Invalid memory pool pointer.
TX_PTR_ERROR	(0x03)	Invalid pointer (NULL) for any destination pointer.

- **Example**

```
TX_BYTE_POOL my_pool;
CHAR *name;
ULONG available;
ULONG fragments;
TX_THREAD *first_suspended;
```

```
ULONG suspended_count;
TX_BYTE_POOL *next_pool;
UINT status;

/* Retrieve information about a the previously created block pool "my_pool." */
status = tx_byte_pool_info_get(&my_pool, &name, &available, &fragments, &first_suspended,
                               &suspended_count, &next_pool);
/* If status equals TX_SUCCESS, the information requested is valid. */
```

3.4. Overwriting Memory Blocks

It is very important to ensure that the user of allocated memory does not write outside its boundaries. If this happens, corruption occurs in an adjacent (usually subsequent) memory area. The results are unpredictable and quite often fatal!

3.5. Release Memory Allocated from Memory Byte Pools

3.5.1. Overview

Customers should release the memory allocated from memory byte pools in time, according to their memory management requirement. After a memory fragment being released, it will not be merged together with the other free memory fragments until a subsequent allocation search for a large enough memory block.

3.5.2. ThreadX API tx_byte_release

- **Prototype**

```
UINT tx_byte_release(VOID *memory_ptr)
```

- **Description**

This service releases a previously allocated memory area back to its associated pool. If there are threads suspended waiting for memory from this pool, each suspended thread is given memory and resumed until the memory runs out or there are no more suspended threads. The process of allocating memory to suspended threads always begins with the first thread suspended. And the application must prevent using memory area after it is released.

- **Input Parameters**

memory_ptr:
Pointer to the previously allocated memory area.

- **Return Values**

TX_SUCCESS	(0x00)	Successful memory release.
TX_PTR_ERROR	(0x03)	Invalid memory area pointer.
TX_CALLER_ERROR	(0x13)	Invalid caller of this service.

- **Example**

```
unsigned char *memory_ptr;
UINT status;

/* Release a memory back to my_pool. Assume that the memory area was previously allocated from
my_pool. */
status = tx_byte_release((VOID *) memory_ptr);
/* If status equals TX_SUCCESS, the memory pointed to by memory_ptr has been returned to the pool. */
```

3.6. Delete Memory Byte Pools

3.6.1. ThreadX API tx_byte_pool_delete

- **Prototype**

```
UINT tx_byte_pool_delete(TX_BYTE_POOL *pool_ptr)
```

- **Description**

This service deletes the specified memory pool. All threads suspended waiting for memory from this pool are resumed and given a *TX_DELETED* return status. It is the application's responsibility to manage the memory area associated with the pool, which is available after this service completes.

In addition, the application must prevent use of a deleted pool or memory previously allocated from it.

- **Input Parameters**

pool_ptr:
Pointer to a previously created memory pool.

- **Return Values**

TX_SUCCESS	(0x00)	Successful memory pool deletion.
TX_POOL_ERROR	(0x02)	Invalid memory pool pointer.
TX_CALLER_ERROR	(0x13)	Invalid caller of this service.

- **Example**

```
TX_BYTE_POOL my_pool;
```

```
UINT status;

/* Delete entire memory pool. Assume that the pool has already been created with a call to
tx_byte_pool_create. */
status = tx_byte_pool_delete(&my_pool);
/* If status equals TX_SUCCESS, memory pool is deleted. */
```

4 Appendix A References

Table 1: Terms and Abbreviations

Abbreviation	Description
API	Application Programming Interface
ISR	Interrupt Service Routines
LPWA	Low-Power Wide-Area
RAM	Random Access Memory