MediaTek

Maui System Service User Manual

Interrupt Handler and KAL Programming Guide

Document Number:

Preliminary Information

Revision: 1.6

Release Date: September 28, 2006

Legal Disclaimer

Revision History

Revision	Date (mm/dd/yyyy)	Author	Comments
0.1	10/21/2004	Shalyn Chua	Initially integrated version.
0.2	11/09/2004	Shalyn Chua	Adding interrupt controller related APIs.
0.3	05/27/2005	CC Hwang	Fix typo in description of new_evshed API
0.4	07/29/2005	Shalyn Chua	Create new API, msg_send_ext_queue_to_head( ).
0.5	10/25/2005	Shalyn Chua	Create new API, EXTRA_EINT_Registration()
0.6	10/27/2005	Shalyn Chua	Modify EMI customization for 05B and later software package.
0.7	10/28/2005	Shalyn Chua	Correct typo on API EXTRA_EINT_Registration.
0.8	12/06/2005	Shalyn Chua	Remove EMI Customization, please refer to MTK GSM_GPRS_System_Configuration document.
0.9	12/14/2005	Karen Hsu	Add new description to free_local_para, free_peer_buff, hold_peer_buff, hold_local_para for re-entrantable new API / MACRO
1.0	03/02/2006	CC Hwang	Modify new_evshed API (support max delay)
1.1	03/31/2006	Shalyn Chua	Remove EMI customization from the title.
1.2	04/24/2006	CC Hwang	Add one new API kal_adm_check_integrity
1.3	05/10/2006	Shalyn Chua	Adding the maximum timeout period of KAL timer and stack timer.
1.4	06/01/2006	CC Hwang	Add EXT_ASSERT_DUMP() and modify kal_adm_check_integrity()
1.5	07/17/2006	Eddic Hsien	Add limitation descriptions for all resource related create/init APIs. Remove the APIs: kal_delete_timer, stack_deinit_timer.
1.6	09/28/2006	CC Hwang	Support zero-initialization of local parameter.

Table of Contents

Interrupt Handler

External Interrupts
nIRQ
nFIQ
APIs

1 Introduction of External Interrupt Handler

MTK base-band chips provide external interrupt (EINT) for the interrupt triggering from external device. This document subjects to give the processing flow, customization and interface of external interrupt interface.

1.1 Overview

Table 2-1 gives the total number of EINT offered on the MTK base-band chips.

Number of channels	MT6205B	MT6217	MT6218	MT6219
External Interrupt (EINT)	3	4	4	4
Dual mode interrupt (nIRQ)	1	1	1	1
Total number	4	5	5	5

Table 2-1. Total number of external interrupts channels

Features of EINT as below,

(1) Selectable edge or level sensitivity,

(2) Selectable negative or positive polarity

(3) Flexible de-bounce time, in terms of 32KHz; the maximal de-bounce time is 64ms (2048 x 31.25µs).

Features of nIRQ,

(1) It is a dual-mode GPIO, acts as interrupt source if it is configured as nIRQ signal,

(2) Configurable as edge sensitivity with active LOW or level sensitivity with active LOW.

Items	MT6205B	MT6217	MT6218	MT6219
Interrupt code	EINT	EINT	EINT	EINT
	10	11	11	11
	nIRQ	nIRQ	nIRQ	nIRQ
	15	18	18	18
	nFIQ	nFIQ	nFIQ	nFIQ
	X	0	0	0
GPIO number for nIRQ	GPIO 21	GPIO 41	GPIO 41	GPIO 41
GPIO number for nFIQ	X	GPIO 42	GPIO 42	GPIO 42

Table 2-2. Interrupt code on a series of MTK base-band chips

2 Internal Design of External Interrupts

Both EINT and nIRQ are central controlled by system service. EINT channel within range could be registered through **EINT_Registration( )**, repetitive registration is allowed, but only the latest will take effect. However, system could have at most one nIRQ source; once it is registered, all the registry following will be blocked with fatal error “re-register nIRQ HISR” (error code 1 = 0x218, error code 2 = 0).

To protect the system from the intervention of instable external interrupt, software de-bouncing time is applied to provide further protection. Sub-sections following illustrate the software de-bouncing, EINT and nIRQ processing flow in detailed.

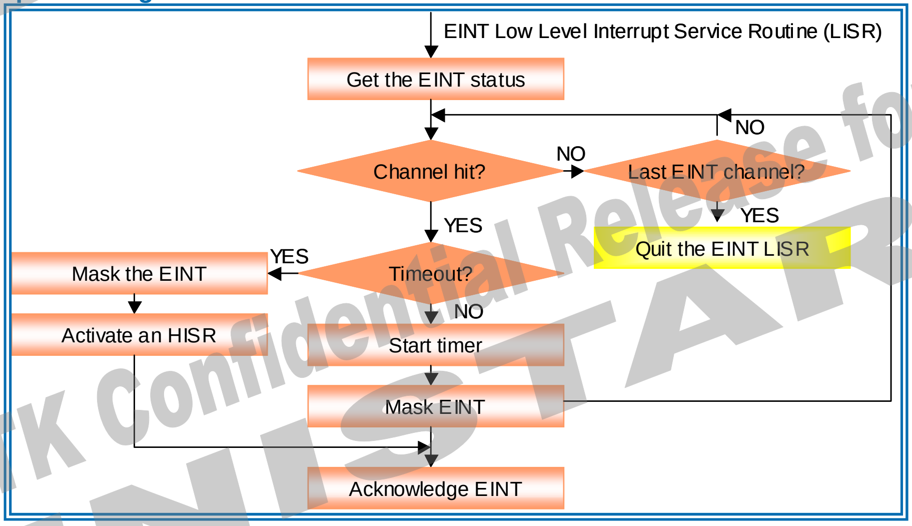
2.1 Software De-bouncing

In addition to hardware embedded de-bounce time 64ms (2048 x 31.25μs), a software de-bounce time is embedded on each EINT channel. Any level triggered interrupt should hold the state over the pre-defined software de-bounce time, otherwise, its callback function would not be activated.

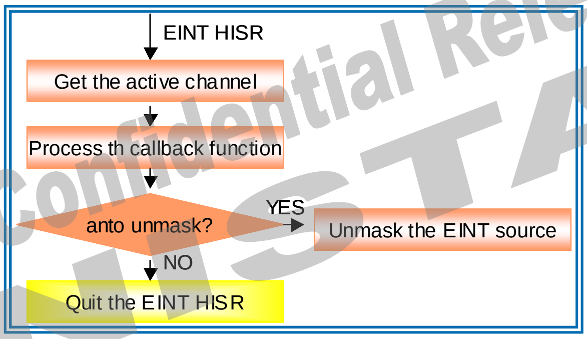
The software de-bounce time are customizable at ***mcu\custom\drv\misc_drv\board version\eint_def.c***, entries of array **custom_eint_sw_debounce_time_delay** is consistent with total number of EINT channels. The setting is in units of 10ms.

```
kal_uint8 custom_eint_sw_debounce_time_delay[EINT_MAX_CHANNEL] =
{
  50,   /*EINT 0,500ms*/
  50,   /*EINT 1,500ms*/
  50    /*EINT 2,500ms*/
};
```

2.2 Internal processing flow of EINT



Processing flow of EINT LISR

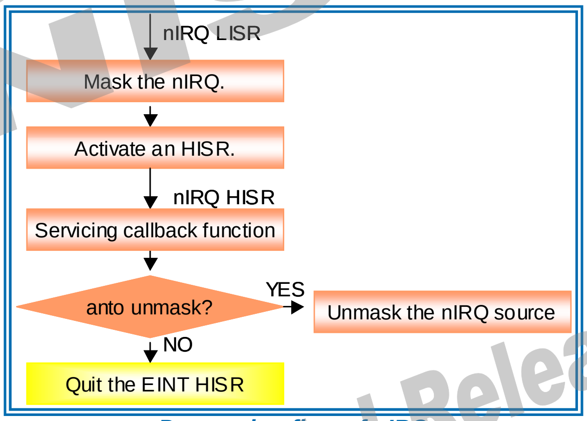


Processing flow of EINT HISR

On the very first happening of an EINT LISR, system won't take action, but start a timer according to the pre-defined software de-bouncing time instead. Meanwhile, EINT source is disabled or masked until the timeout routine is serviced. The next occurrence of the same EINT LISR will then activate the EINT HISR, and EINT callback function registered through **EINT_Registration( )** takes effect finally. However, if the external interrupt signal no longer holds, the EINT interrupt status will be reset internally, and kept unmasked for the next issuing.

At the HISR (high level interrupt service routine), EINT unmasking will be done according to the user's specification. System handles the EINT unmasking if auto_unmask is set to KAL_TRUE.

2.3 Internal processing flow of nIRQ



Processing flow of nIRQ

It is different from EINT that, no de-bouncing is done on software level; the very first LISR will be serviced and the correspondence callback function is done at HISR level.

3 API

Below are APIs for interrupts controller, EINT and nIRQ.

3.1 Interrupt Controller

IRQMask

Prototype:	void IRQMask(kal_uint8 no)
Header file:	intrCtrl.h
Input:	*no* is interrupt source to be disabled, please refer to datasheet of related base-band chip for number and index of interrupt sources.
Description:	This function serves for disabling dedicated interrupt.

IRQUnmask

Prototype:	void IRQUnmask(kal_uint8 no)
Header file:	intrCtrl.h
Input:	*no* is interrupt source to be enabled, please refer to datasheet of related base-band chip for number and index of interrupt sources.
Description:	This function serves for enabling dedicated interrupt.

SaveAndSetIRQMask

Prototype:	kal_uint32 SaveAndSetIRQMask(void)
Header file:	intrCtrl.h
Output:	Current value of CPSR.
Description:	This function is special for disabling all interrupt sources by setting I-bit, besides, return the current value of CPSR. It must be used in-paired with **RestoreIRQMask( )**.

RestoreIRQMask

Prototype:	void RestoreIRQMask(kal_uint32 value)
Header file:	intrCtrl.h
Input:	*value* is CPSR value to be restored.
Description:	Restore the I-bit once it is turned-off. It must be used in-paired with **SaveAndSetIRQMask( )**.

Example:

```
kal_uint32 CriticalFunction(void)
{
        kal_uint32 savedMask;

        /* I-bit should be turned-off at this critical path */
        savedMask = SaveAndSetIRQMask();
        ♦♦♦♦♦♦♦♦
         RestoreIRQMask(savedMask);
}
```

3.2 EINT

EINT_Registration

Prototype: void EINT_Registration (kal_uint8 eintno, kal_bool Dbounce_En, kal_bool ACT_Polarity, void (reg_hisr)(void), kal_bool auto_umask)

Header file: eint.h

Input: *eintno* is channel of EINT to be registered, *Dbounce_En* tells if hardware de-bounce time need to be enabled, *ACT_Polarity* is the active polarity, *reg_hisr* is callback function and *auto_umask* tells the system if it needs to unmask the interrupt at end of processing.

Description: This service provides the registry function for EINT, any illegal EINT number will be rejected with an ASSERT. It is suggested that, always keep the hardware de-bouncing active; hardware issues the EINT interrupt after 64ms de-bouncing. Any EINT interrupt is either positive (KAL_TRUE) or negative (KAL_FALSE) level trigger, and user should also specify the callback function as well as auto-unmask flag.

EXTRA_EINT_Registration

Prototype: void EXTRA_EINT_Registration (kal_uint8 eintno, kal_bool ACT_Polarity, void (reg_hisr)(void), kal_bool auto_umask)

Header file: eint.h

Input: *eintno* is channel of EINT to be registered, *ACT_Polarity* is the active polarity, *reg_hisr* is callback function and *auto_umask* tells the system if it needs to unmask the interrupt at end of processing.

Description: Some MTK BB-chips provides 4 external interrupts with hardware de-bounce supported; besides, configurable level or edge trigger external interrupts. In addition to the 4 external interrupt source, we have additional external interrupts multiplexed with UART2 RX, UART2 TX, UART3 RX and UART3 TX, which also could be served as external interrupt sources without hardware de-bounce capability and always level trigger.
This service specific for the registration of these additional 4 external interrupts. Similar with EINT_Registration, any illegal EINT number will be rejected with an ASSERT. Any EINT interrupt is either positive (KAL_TRUE) or negative (KAL_FALSE) level trigger, and user should also specify the callback function as well as auto-unmask flag.

Availability: W05.36 and later.

EINT_Set_Polarity

Prototype: void EINT_Set_Polarity (kal_uint8 eintno, kal_bool ACT_Polarity)

Header file: eint.h

Input: *eintno* is the destination EINT channel, *ACT_Polarity* is new polarity to be set.

Description: This function is provided to set the polarity accordingly, KAL_TRUE for positive level trigger and KAL_FALSE for negative level trigger. The whole procedure is done under the protection of disabling interrupt.

EINT_Mask

Prototype: void EINT_Mask (kal_uint8 eintno)

Header file: eint.h

Input: *eintno* is the destination EINT channel.

Description: This function is provided for masking/disabling the given EINT channel. Again, the action is done under the protection of disabling interrupt.

EINT_UnMask

Prototype:	void EINT_UnMask (kal_uint8 eintno)
Header file:	eint.h
Input:	*eintno* is the destination EINT channel.
Description:	This function is provided for unmasking/enabling the given EINT channel. Again, the action is done under the protection of disabling interrupt.

EINT_SW_Debounce_Modify

Prototype:	kal_int32 EINT_SW_Debounce_Modify (kal_uint8 eintno, kal_uint8 debounce_time)
Header file:	eint.h
Input:	*eintno* is the destination EINT channel, *debounce_time* is the new de-bounce time.
Output:	-1 if the given EINT channel is illegal, 1 if the function is completely done.
Description:	This function call aims at dynamically modifying de-bouncing time of the given EINT channel. Note that, the initial de-bounce time is specified at **custom_eint_sw_debounce_time_delay**, but it could be adjusted at run-time. This function is also done under the protection of disabling interrupt.

3.3 nIRQ

nIRQ_init

Prototype:	kal_bool nIRQ_init (void)
Header file:	isrentry.h
Output:	KAL_TRUE if operation successfully done, otherwise KAL_FALSE will be returned.
Description:	This service aims at configuring GPIO mode; please refer to Table 2-2 for the GPIO mapping.
Remark:	**It must be called after Drv_Init(), which is called from Application_Initialize(), Initialize(); otherwise, the value will be overwritten.**

nIRQ_Registration

Prototype:	void nIRQ_Registration (kal_bool edge, void(reg_hisr)(void), kal_bool auto_unmask)
Header file:	isrentry.h
Input:	*edge* is the polarity, *reg_hisr* is callback function and *auto_umask* tells the system if it needs to unmask the interrupt and end of processing.
Description:	This service provides the registry function for nIRQ, it is either falling edge trigger (KAL_TRUE) or negative level trigger (KAL_FALSE), and user should also specify the callback function and auto-unmask flag.

3.4 nFIQ

nFIQ_Init

Prototype:	kal_bool nFIQ_init (void (hisr_callback)(void), kal_bool enable, kal_bool auto_unmask, kal_bool sensitivity)
Header file:	isrentry.h
Input:	*hisr_callback* is the callback function of nFIQ HISR. If *enable* is KAL_TRUE, system will enable nFIQ at nFIQ_Init. *auto_unmask* leads the automatically unmask of nFIQ at HISR, otherwise, user is responsible to unmask it. *sensitivity* is used to specify the sensitivity, KAL_FALSE tells LEVEL_SENSITIVE, and KAL_TRUE is EDGE_SENSITIVE.

Output:	Apply this function on base-band chip without nFIQ results in the return value KAL_FALSE, otherwise KAL_TRUE.
Description:	This service aims at configuring GPIO mode; please refer to Table 2-2 for the GPIO mapping. Besides, LISR registration and HISR creation of nFIQ are also done in the function.
Remark:	**(1) It must be called after Drv_Init(), which is called from Application_Initialize(), Initialize(); otherwise, the value will be overwritten.** **(2) Users must be very careful in using nFIQ, because it is the highest priority interrupt by default.**

KAL Programming Guide

Internal Design

Data Types

Data Structure

APIs

Examples

4 KAL – An Overview

KAL, abbreviation of Kernel Adaptation Layer, is an adaptation layer between Operating System (OS) and upper layer applications. To keep it highly portable, KAL defines its own API set for each OS component, including task management, task synchronization, task communication, timer management and memory management.

Being a robust adaptation layer, KAL is rich of amazing features,

- Entry of OS functions,
- System call parameter checking,
- System abruption tracking,
- Tracking the peak consumption of system resource.

Among them, debugging and profiling related features are well protected with compile option, and could be optionally turned-off if they are no longer needed.

Section 2 is general description of data types defined in KAL, illustration of internal flow and API of each abovementioned component are provided in subsequent sections.

Notations:

Black, bolded and italic wording is either a ***file name*** or ***file name with relative path***.
Black, bolded and underlined wording is a **<u>variable</u>**.
Black, bolded, underlined and quoted wording is a **<u>function( )</u>**.
Bolded and blue color wording is **structure name.**

5 Fundamental Data Types

Table below summarizes all the fundamental data types provided by KAL, they work if ***kal_release.h*** is included. Components abbreviated as NU_xxx are Nucleus Plus core related, please refer to Nucleus Plus documentation.

Data Types	Description	Data Types	Description
kal_char	Type of character.	kal_os_task_type	Equivalent to NU_TASK
kal_int8	8bits signed-integer.	kal_os_hisr_type	Equivalent to NU_HISR
kal_uint8	8bit unsigned-integer.	kal_os_queue_type	Equivalent to NU_QUEUE
kal_int16	16bits signed-integer.	kal_os_mutex_type	Equivalent to NU_SEMAPHORE
kal_uint16	16bit unsigned-integer.	kal_os_sem_type	Equivalent to NU_SEMAPHORE
kal_int32	32bits signed-integer.	kal_os_eventgrp_type	Equivalent to NU_EVENT_GROUP
kal_uint32	32bit unsigned-integer.	kal_os_timer_type	Equivalent to NU_TIMER
kal_int64	64bits signed-integer.	kal_os_pool_type	Equivalent to NU_PARTITION_POOL
kal_uint64	64bit unsigned-integer.	WCHAR	16bit unsigned short.
kal_bool	Boolean type typedef enum { KAL_FALSE, KAL_TRUE } kal_bool;	kal_wait_mode	Waiting style. typedef enum { KAL_NO_WAIT, KAL_INFINITE_WAIT } kal_wait_mode;
kal_status	typedef enum { KAL_SUCCESS, KAL_ERROR, KAL_Q_FULL, KAL_Q_EMPTY, KAL_SEM_NOT_AVAILABLE, KAL_WOULD_BLOCK, KAL_MESSAGE_TOO_BIG, KAL_INVALID_ID, KAL_NOT_INITIALIZED, KAL_INVALID_LENGHT, KAL_NULL_ADDRESS, KAL_NOT_RECEIVE, KAL_NOT_SEND, KAL_MEMORY_NOT_VALID, KAL_NOT_PRESENT, KAL_MEMORY_NOT_RELEASE } kal_status;	KAL_ADM_ID	Type of void *.

Table 6-1. Fundamental data types

6 Task Management

Task management unit is in-charge of the creation of system and customer-defined tasks, besides, scheduling the execution of tasks and High Level Interrupt Service Routine (HISR).

6.1 Description

Basic execution unit on MAUI is either a task or HISR, and could be interrupted. Context switch takes place no matter system call is trapped or interrupts; and scheduling scheme applied on the system is priority scheduling. Table below states the priority coverage and their distribution convention. Any two tasks or two HISRs, which have identical priority, are scheduled in sequential manner.

Execution Unit	Priority	Description
HISR	0	The highest priority HISR, is always reserved for L1_HISR, others are prohibited! Otherwise, fatal error would be encountered during HISR creation. (Fatal error code 1 = 0x213, code 2 = 0x04)
	1	The second highest priority HISR.
	2	The lowest priority HISR, however, it still takes priority than tasks.
Task	KAL_PRIORITY_CLASS0 ~ (KAL_PRIORITY_CLASS18)	Reserved for system usage only, *it is suggested that customer tasks must not use the priority within the range.*
	(KAL_PRIORITY_CLASS18 + 1) ~ (KAL_PRIORITY_CLASS19 + 9)	For tasks, which are timing critical, like BMT (Battery Management Task), AUX and OBEX.
	KAL_PRIORITY_CLASS20 ~ (KAL_PRIORITY_CLASS21+9)	Applications like MMI, WAP and JAVA occupy priority within the range.
	KAL_PRIORITY_CLASS22 ~ (KAL_PRIORITY_CLASS24 + 9)	For tasks, which have rather lower priority, and to be scheduled when the system is free, for instance, NVRAM and TST tasks.
	KAL_PRIORITY_CLASS25 ~ (KAL_PRIORITY_CLASS25 + 5)	For very low priority tasks, for example, priority (KAL_PRIORITY_CLASS25 + 5) is reserved of IDLE task.

Table 7-1. Convention of priority coverage

There are three possible boot-modes, META, USB or normal boot mode. Different tasks are created for differenct boot-mode, they are pre-defined in a constant array of type **comptask_info_struct**, namely **sys_comp_config_tbl** for normal boot mode; on the other hand, customer-defined tasks are defined in **custom_comp_config_tbl**. At booting stage, system is responsible to identify the exact boot mode, furthermore, copy the tasks information onto global array known as **task_info_g**, which is an important reference pool in run-time task management. For instance, in message passing, **task_info_g** will be always be referenced to target the destination queue ID.

A task is in READY state soon after creation, or ready to be executed; and system schedules the highest priority task at the very first context switch. During execution period, it may be suspended for resource synchronization or communication. Table 7-2 summarizes all possible task status.

Status	Notation	Description
NU_READY	0	Task is in ready state, ready to be scheduled and executed.
NU_SLEEP_SUSPEND	2	Task is in sleeping state by calling kal_sleep_task().
NU_QUEUE_SUSPEND	4	Task is suspended at queue, will be awaken in case of messages arrival.
NU_SEMAPHORE_SUSPEND	6	Task is suspended at semaphore, will return to READY state if semaphore becomes available.
NU_EVENT_SUSPEND	7	Task is suspended and waiting the validity of an event.

Table 7-2. Task status

Sub-sections following show you the data structure and data type widely used in task management unit, as well as task management related APIs.

6.2 Data Structures and Data Types

Tables following illustrate the general data structures and data types applied on task management unit.

comptask_handler_struct *(mcu\config\include\syscomp_config.h)*

Description:

Defines the task configuration component, including task entrance, initialization, and others three optional functions, configuration, reset and end functions.

Data Type	Element	Description
kal_task_func_ptr	comp_entry_func	Task entry function.
task_init_func_ptr	comp_init_func	Task initialization function (optional).
task_cfg_func_ptr	comp_cfg_func	Task configuration function (optional).
task_reset_func_ptr	comp_reset_func	Task reset function (optional).
task_end_func_ptr	comp_end_func	Task end function (optional).

comptask_info_struct *(mcu\config\include\syscomp_config.h)*

Description:

Constant type array, it is used for task description, the task would not be created if its create function is NULL.

Data Type	Element	Description
kal_char *	comp_name_ptr	Task name.
kal_char *	comp_qname_ptr	Name of external queue.
kal_uint32	comp_priority	Task priority.
kal_uint16	comp_stack_size	Stack size in terms of Bytes.
kal_uint8	comp_ext_qsize	Number of external queue entries.
kal_uint8	comp_int_qsize	Number of internal queue entries.
kal_create_func_ptr	comp_create_func	Function pointer of creates function, with comptask_handler_struct as input argument, and return type is kal_bool.
kal_bool	comp_internal_ram_stack	Specify if the stack size to be created from internal SRAM, *it is not suggested that customers create their tasks' stack at internal SRAM.*

task_info_struct *(mcu\config\include\task_config.h)*

Description:

Keep the run-time task related information,

Data Type	Element	Description
kal_char *	task_name_ptr	Pointer of task's name, reference from element comp_name_ptr of **comptask_info_struct**.
kal_char *	task_qname_ptr	Name of external queue, reference from element comp_qname_ptr of **comptask_info_struct**.
kal_uint32	task_priority	Task priority, copied from comp_priority of **comptask_info_struct.**
Kal_uint16	task_stack_size	Stack size in terms of bytes, also duplicated from comp_stack_size of **comptask_info_struct**.

kal_taskid	task_id	Task ID, which is assigned by OS at run-time.
kal_msgqid	task_ext_qid	Task's external queue ID, which is assigned by OS after creation.
int_q_type	*task_int_qid_ptr	Task's internal queue ID, which is assigned by OS after creation.
kal_task_func_ptr	task_entry_func	Task's entry function duplicated from comp_create_func of **comptask_info_struct**.
task_cfg_func_ptr	task_cfg_func	Task's configuration function duplicated from comp_create_func of **comptask_info_struct**.
task_init_func_ptr	task_init_func	Task's initialization function duplicated from comp_create_func of **comptask_info_struct**.
task_reset_func_ptr	task_reset_func	Task's reset function duplicated from comp_create_func of **comptask_info_struct**.
task_end_func_ptr	task_end_func	Task's end function duplicated from comp_create_func of **comptask_info_struct**.
kal_uint8	task_ext_qsize	Total number of external queue entries copied from comp_ext_qsize of **comptask_info_struct**.
kal_uint8	task_int_qsize	Total number of internal queue entries copied from comp_int_qsize of **comptask_info_struct**.
kal_bool	task_internal_ram_stack	If internal SRAM (__SYS_INTERN_RAM__) is defined, a task stack could be selectively created at internal or external SRAM. *It is not suggested that customers create their tasks' stack at internal SRAM.*

kal_task_type, *kal_internal_taskid ***(mcu\kal\nuclues\include\kal_nucleus.h)***		
Description: Defines the task control block, slightly different if DEBUG_KAL is defined (light yellow background color).		
Data Type	**Element**	**Description**
kal_os_task_type	task_id	Inherits from Nucleus Plus task control block.
kal_char *	task_name	Name of the task.

kal_hisr_type, *kal_internal_hisrid ***(mcu\kal\nuclues\include\kal_nucleus.h)***		
Description: Defines the HISR control block, slightly different if DEBUG_KAL is defined (light yellow background color).		
Data Type	**Element**	**Description**
kal_os_hisr_type	task_id	Inherits from Nucleus Plus HISR control block.
kal_char *	hisr_name	Name of the HISR.

Data Types	**Description**
task_indx_type	Enumeration type of task ID.
module_type	Enumeration type of module ID.
kal_taskid	Identity of a task, it is internally equivalent to kal_internal_taskid.
kal_hisrid	Identity of a HISR, it is internally equivalent to kal_internal_hisrid.
task_entry_struct	Structure of single element, task_indx_type.
kal_task_func_ptr	Function pointer for task entry function, with task_entry_struct * as input argument.

task_init_func_ptr	Function pointer for task initialization function with task_indx_type as input argument, and output type is kal_bool.
task_end_func_ptr	Function pointer for task end function with task_indx_type as input argument, and output type is kal_bool.
task_reset_func_ptr	Function pointer for task reset function with task_indx_type as input argument, and output type is kal_bool.
task_cfg_func_ptr	Function pointer for task configuration function with task_indx_type as input argument, and output type is kal_bool.
kal_create_func_ptr	Function pointer with comptask_handler_struct ** as input argument, and return type kal_bool, which is used for task creation function.

Global variable	Description
task_info_g	Array of **task_info_struct,** number of the entries is determined by enum of task ID plus 16 customer-defined tasks.

6.3 Task Management API

kal_activate_hisr

Prototype: void kal_activate_hisr (kal_hisrid ext_hisr_id)

Header file: kal_release.h

Input: *hisrid* is destination HISR to be activated.

Description: This function call is widely used to activate an HISR; once it is activated, system will pick up the highest priority HISR in the next scheduling.

Example:

```
static kal_hisrid dma_hisr;

void DMA_LISR(void)
{
  IRQMask(IRQ_DMA_CODE);
  kal_activate_hisr(dma_hisr);
```

kal_change_priority

Prototype: kal_uint32 kal_change_priority (kal_taskid taskid, kal_uint32 new_priority)

Header file: kal_release.h

Input: *taskid* is destination task, *new_priority* is new priority number to be assigned.

Output: Old priority setting.

Description: This function offers run-time task priority switching, users must be very careful in using the function, otherwise, deadlock may occur due to priority change.

kal_create_hisr

Prototype: kal_hisrid kal_create_hisr (kal_char* hisr_name, kal_uint8 priority, kal_uint32 stack_size, kal_hisr_func_ptr entry_func, kal_uint8 options)

Header file: kal_release.h

Input: *hisr_name* is name of HISR to be created, *priority* tells the HISR priority, either 1 or 2.; *stack_size* is stack size associates with this HISR, which is in terms of bytes, if *options* is KAL_FALSE, stack will be allocated from internal SRAM, otherwise from external SRAM. Only timing critical HISR is suggested to put its stack at internal SRAM. Finally, *entry_func* tells the HISR entry function.

Output: ID of the HISR.

Description: Function used to create an HISR. **It is strongly suggested to call this function only at system initialization stage, and the HISR related data elements could not be freed once it's created.**

Remark: *Priority number 0 is strictly prohibited, which is specially reserved for the highest priority HISR.*

kal_get_my_task_index

Prototype: void kal_get_my_task_index (kal_uint32 *index)

Header file: kal_release.h

Input: Pointer of type unsigned 32 bits, which is used for return value.

Description: Finding out correspondence task index of currently running execution unit; if it is not a task, *index* would be total number of tasks –1.

kal_get_mytask_priority

Prototype:	kal_uint32 kal_get_mytask_priority (void)
Header file:	kal_release.h
Output:	Pointer of type unsigned 32 bits, which is used for return value.
Description:	Retrieving priority of current execution unit, either a task or an HISR; for the former, return value ranges from 0 to 255, while the later is either 0, 1 or 2.

kal_get_task_self_id

Prototype:	kal_taskid kal_get_task_self_id (void)
Header file:	kal_release.h
Output:	Current task ID.
Description:	Return the current executing task ID, if the current execution unit is not a task, KAL_NILTASK_ID will be returned.

kal_if_hisr

Prototype:	kal_bool kal_if_hisr (void)
Header file:	kal_release.h
Output:	KAL_FALSE if the current execution unit is a task, and KAL_TRUE if the current execution unit is a HISR.
Description:	Identify if the current execution unit is a HISR.

kal_sleep_task

Prototype:	void kal_sleep_task (kal_uint32 time_in_ticks)
Header file:	kal_release.h
Input:	*time_in_ticks* is sleeping duration, each unit is 4.615ms.
Description:	Forcing a task to sleep for a duration, which is expressed in terms of 4.615ms.

stack_change_priority_by_module_ID

Prototype:	kal_uint32 stack_change_priority_by_module_ID (module_type mod_ID, kal_uint32 new_priority)
Header file:	kal_release.h
Input:	*mod_ID* is module ID of destination task whose priority is going to be adjusted, *new_priority* is new priority number to be assigned.
Output:	Old priority setting.
Description:	This function also offers run-time task priority switching, its input argument is different from **kal_change_priority ( )**, where a task is targeted via module ID rather than task ID. Users must be very careful in using the function, otherwise, deadlock may occur due to priority change.

6.4 Customization

A customer task must be specified in **custom_comp_config_tbl** (defined in ***mcu\custom\system\board version\custom_config.c***), which is an array of type **comptask_info_struct**. Its existence could be identified by create function, NULL if not exist.

In booting stage, system will look into the table, and determine its creation via create function. To keep the consistency, *customers are discouraged to create a task by calling **kal_create_task( )** directly.*

Following is the step-by-step illustration about creating a customer task.

Step 1, Define task ID and module ID

Task ID and module ID are two fundamental identifier of a task; the former is widely used for targeting a task entry, while the later is widely used in message passing. They are defined in ***mcu\custom\system\board version\custom_config.h***.

```
typedef enum {
   INDX_CUSTOM1 = RPS_CUSTOM_TASKS_BEGIN,
   INDX_CUSTOM2,
RPS_CUSTOM_TASKS_END
} custom_task_indx_type;
```

```
typedef enum {
MOD_CUSTOM1 = MOD_CUSTOM_BEGIN,
MOD_CUSTOM2,
MOD_CUSTOM_END
} custom_module_type;
```

The red bolded wordings must not be eliminated, because system relies on them to restrict the number of customer tasks. At most 16 customer defined task ID and 16 customer-defined modules ID are allowable nowadays. In case of violation, fatal error "Customer creates too many tasks" (fatal error code 1 = 0x1501, error code 2 = number of customer-defined task) and fatal error "Customer defines too many module IDs" (fatal error code 1 = 0x1502, error code 2 = number of customer-defined module) will be suffered respectively.

Step 2, Fill-in module to task ID mapping table

Complete the module ID to task ID mapping table, it is defined in ***mcu\custom\system\board version\ custom_config.c***..

Following the previous example, both task INDX_CUSTOM1 and INDX_CUSTOM2 associate with single module ID, therefore, entry MOD_CUSTOM1 in table **custom_mod_task_g** maps to task INDX_CUSTOM1. Also, *INDX_NIL must not be removed*.

```
custom_task_indx_type custom_mod_task_g[ MAX_CUSTOM_MODS ] =
{
  INDX_CUSTOM1,       /* MOD_CUSTOM1 */
  INDX_CUSTOM2,       /* MOD_CUSTOM2 */
  INDX_NIL            /* Please end with INDX_NIL element */
};
```

Step 3, Configure the task creation table

Define the task configuration information in ***mcu\custom\system\board version\custom_config.c,*** please refer to data type Comptask_info_struct for more detail.

```
const comptask_info_struct custom_comp_config_tbl[ MAX_CUSTOM_TASKS ] =
{
  /* INDX_CUSTOM1 */
  {"CUST1", "CUST1 Q", 210, 1024, 10, 0,
#ifdef CUSTOM1_EXIST
  custom1_create, KAL_FALSE},
#else
  NULL, KAL_FALSE},
#endif

  /* INDX_CUSTOM2 */
  {"CUST2", "CUST2 Q", 211, 1024, 10, 0,
#ifdef CUSTOM2_EXIST
  custom2_create, KAL_FALSE},
#else
  NULL, KAL_FALSE},
#endif
};
```

Step 4, Implement the task create function

Implement the task create functions in ***mcu\custom\system\board version\custom1_create.c,*** please refer to data type comptask_handler_struct for more detail.

```
kal_bool custom1_create(comptask_handler_struct **handle)
{
  static const comptask_handler_struct custom1_handler_info =
  {
    custom1_main,  /* task entry function */
    NULL,  /* task initialization function */
    NULL,  /* task configuration function */
    NULL,  /* task reset handler */
    NULL,  /* task termination handler */
  };
  *handle = (comptask_handler_struct *)&custom1_handler_info;
  return KAL_TRUE;
}
```

Step 5, Define the message ID

Define the message ID in ***mcu\custom\system\board version\ custom_sap.h,*** which must be started with "MSG_ID_".

```
/* Add customization message id here */
MSG_ID_CUSTOM1_CUSTOM2  =  CUSTOM_MSG_CODE_BEGIN,
MSG_ID_CUSTOM2_CUSTOM1,
```

Currently, the maximal allowable customer message ID is 1000, fatal error “Customer define too many message IDs” (fatal error code 1 = 0x1503, error code 2 = total number of customer-defined message ID) will be encountered if exceeding.

7 Task Synchronization Management

The simplest form of synchronization involves the starting of a task at the appropriate time. By providing message exchange and wait/wake mechanisms, MAUI is capable of offering a variety of solutions to the problem of task synchronization. Message exchange system is categorized as "Task Communication Management" and will be described in next chapter. This chapter focuses on wait/wake –based mechanism.

7.1 Description

By using event group, a task can trigger an event, someone who is waiting for the event would be waken-up by the Operating System (OS). Also, to protect a critical section, users could rely on MUTEX or semaphore to achieve the goal. MUTEX inherits from semaphore with initial courting value 1. The state of task exactly reveals the suspension type.

7.1.1 Event group

Event group provides a mechanism to indicate that a certain system event has occurred. A single bit in an event group represents an event. This bit is called an event flag. There are 32 event flags in each event group.

Tables below collects the possible operations for event retrieval and event set respectively.

Operations	Description
KAL_AND	Indicate that all of the requested event flags are required.
KAL_AND_CONSUME	Indicate that all of the requested event flags are required, and CONSUME option automatically clears the event flags present on a successful request.
KAL_OR	Indicate that one or more of the requested event flags is sufficient.
KAL_OR_CONSUME	Indicate that one or more of the requested event flag is sufficient, and CONSUME option automatically clears the event flags present on a successful request.

Table 8-1. Operations on retrieving event group

Operations	Description
KAL_AND	Causes the event flags specified to be "ANDed" with the current event flags in the group.
KAL_OR	Causes the event flags specified to be "ORed" with the current event flags in the group. Note: Event flags can be cleared with the NU_AND option.

Table 8-2. Operations on setting event group

7.2 Data structures and Data Types

kal_mutex_stat_type, *kal_internal_mutex_statistics *(mcu\kal\nuclues\include\kal_nucleus.h)*

Description:
This data structure aims at tracking the owner and status of a MUTEX.

Data Type	Element	Description
kal_internal_task_id	owner_task	Task, who owns the MUTEX.
al_mutex_state	mutex_state	State of MUTEX, either in KAL_MUTEX_GIVEN or KAL_MUTEX_TAKEN state.

kal_mutex_type, *kal_internal_mutexid *(mcu\kal\nuclues\include\kal_nucleus.h)*

Description:
Defines the control block of a MUTEX, slightly different if DEBUG_KAL and DEBUG_ITC are defined (light yellow background).

Data Type	Element	Description
kal_os_mutex_type	mutex_id	Inherits from semaphore of Nucleus Plus.
kal_os_task_type *	owner_task	Owner of the MUTEX, which is expressed with task identity.
kal_mutex_state	mutex_state	Status of the MUTEX, either in KAL_MUTEX_GIVEN or KAL_MUTEX_TAKEN.
kal_internal_mutex_statistics	mutex_stat	Pointer to statistical data structure of a MUTEX.

kal_sem_type, *kal_internal_semid *(mcu\kal\nuclues\include\kal_nucleus.h)*

Description:
Defines the control block of a MUTEX, slightly different if DEBUG_KAL and DEBUG_ITC are defined (light yellow background).

Data Type	Element	Description
kal_os_sem_type	sem_id	Inherits from semaphore of Nucleus Plus.
kal_sem_state	sem_state	Status of the semaphore, either in KAL_SEM_GIVEN or KAL_SEM_TAKEN.

Data Types	Description
kal_mutex_state	Status of a MUTEX, KAL_MUTEX_GIVEN if it is in given state, or KAL_MUTEX_TAKEN if it is in TAKEN state.
kal_mutexid	Identity of a MUTEX, which is internally equivalent to kal_internal_mutexid.
kal_sem_state	Status of a semaphore, KAL_SEM_GIVEN if it is in given state, or KAL_SEM_TAKEN if it is in TAKEN state.
kal_semid	Identity of a semaphore, which is internally equivalent to kal_internal_semid.
kal_eventgrp_type	Alias of NU_EVENT_GROUP.
kal_internal_eventgrpid	Pointer of type NU_EVENT_GROUP, which is used for internal processing.
kal_eventgrpid	Identity of an event group, which is internally equivalent to kal_internal_eventgrpid.

7.3 Task Synchronization APIs

7.3.1 MUTEX

kal_create_mutex

Prototype: kal_mutexid kal_create_mutex (kal_char* mutex_name)
Header file: kal_release.h
Input: *mutex_name* is name of the MUTEX to be created.
Output: Pointer of the created MUTEX.
Description: This function call is dedicated for creating a MUTEX. **It is strongly suggested to call this function only in system initialization time, and the related data allocated for the MUTEX could not be freed once it's created.**

kal_take_mutex

Prototype: void kal_take_mutex (kal_mutexid ext_mutex_id_ptr)
Header file: kal_release.h
Input: *ext_mutex_id_ptr* is destination MUTEX to be taken.
Description: This service obtains an instance of the specified MUTEX. If the MUTEX is in KAL_MUTEX_TAKEN state before this call, the service cannot be immediately satisfied, and caller will be suspended endlessly. Once the MUTEX is taken by a task, its status will soon become KAL_MUTEX_TAKEN, and owner ID will be recorded in control block of the MUTEX.

kal_give_mutex

Prototype: void kal_give_mutex (kal_mutexid ext_mutex_id_ptr)
Header file: kal_release.h
Input: *ext_mutex_id_ptr* is destination MUTEX to be given.
Description: This service releases an instance of the MUTEX specified by the parameter *ext_mutex_id_ptr*. If there are any tasks waiting to obtain the same MUTEX, the first task waiting is given this instance of the MUTEX. Otherwise, if there are no tasks waiting for this MUTEX, the internal counter is incremented by one. Soon after giving the MUTEX, the status will be updated as KAL_MUTEX_GIVEN. Also, in case of a MUTEX is not given and taken by the same task, fatal error "kal_give_mutex: The mutex is taken by another task" (fatal error code 1 = 0x405, error code 2 = *ext_mutex_id_ptr*), will be taken place.

7.3.2 Semaphore

kal_create_sem

Prototype: kal_semid kal_create_sem (kal_char* sem_name, kal_uint32 initial_count)
Header file: kal_release.h
Input: *sem_name* is name of the semaphore to be created, whose initial value is stated in *initial_count*.
Output: Pointer of the created semaphore.
Description: This service creates a counting semaphore, semaphore values can range from 0 through 4,294,967,294. **It is strongly suggested to call this function only in system initialization time, and the related data allocated for the semaphore could not be freed once it's created.**

kal_take_sem

Prototype: kal_status kal_take_sem (kal_semid ext_sem_id_ptr, kal_wait_mode wait_mode)

Header file: kal_release.h
Input: *ext_mutex_id_ptr* is destination semaphore to be taken, and *wait_mode* aims for specifying the waiting style, KAL_INFINITE_WAIT triggers endless suspension, otherwise KAL_NO_WAIT.
Output: KAL_SUCCESS if the operation is done successfully; KAL_SEM_NOT_AVAILABLE if the semaphore is unavailable.
Description: This service obtains an instance of the specified semaphore. Since the instances are implemented with an internal counter, obtaining a semaphore translates into decrementing the semaphore's internal counter by one. If the semaphore counter reaches zero before this call, the service cannot be immediately satisfied. The waiting style determines if a task waits infinitely or returns immediately.

kal_give_sem

Prototype: void kal_give_sem (kal_semid ext_sem_id_ptr)
Header file: kal_release.h
Input: *ext_sem_id_ptr* is destination semaphore to be given.
Description: This service releases an instance of the semaphore specified by the parameter *ext_sem_id_ptr*. If there are any tasks waiting to obtain the same semaphore, the first task waiting is given this instance of the semaphore. Otherwise, if there are no tasks waiting for this semaphore, the internal semaphore counter is incremented by one.

7.3.3 Event group

kal_create_event_group

Prototype: kal_eventgrpid kal_create_event_group (kal_char* eventgrp_name)
Header file: kal_release.h
Input: *eventgrp_name* is name of the event group to be created.
Output: Pointer of the created event group.
Description: This service creates an event flag group, each event flag group contains 32 event flags. All event flags are initially set to 0.

kal_set_eg_events

Prototype: kal_status kal_set_eg_events (kal_eventgrpid eg_id, kal_uint32 events, kal_uint8 operation)
Header file: kal_release.h
Input: *eg_id* is destination event group to be set, and *events* are event flags to be set, while *operation* defines various operation mode, either KAL_OR or KAL_AND.
Output: KAL_SUCCESS if the operation is done successfully.
Description: This service sets the specified event flags in the specified event group. Any task waiting on the event group whose event flag request is satisfied by this service is resumed.

kal_retrieve_eg_events

Prototype: kal_status kal_retrieve_eg_events (kal_eventgrpid eg_id, kal_uint32 requested_events,
kal_uint8 operation, kal_uint32 *retrieved_events, kal_uint32 suspend)
Header file: kal_release.h
Input: *eg_id* is pointer to the user-supplied event flag group control block; *requested_events* is a set bit indicates the corresponding event flag is requested; *operation* specifies the operation options, *retrieved_events* contains event flags actually retrieved, and *suspend* specifies whether to suspend the calling task if the requested event flag combination is not met.

Description: This service retrieves the specified event-flag combination from the specified event-flag group. If the combination is present, the service completes immediately.

7.4 Example

Below is an example of the usage of event group.

```
#define SET_BIT0  0x00000001
#define SET_BIT1  0x00000002
#define SET_BIT2  0x00000004

kal_hisrid        SAMPLE_Hisr;
kal_eventgrpid SAMPLE_Events;

void SAMPLE_HISR_Entry (void)
{
       ♦♦♦♦♦♦♦♦
       /* An HISR sets an event */
         kal_set_eg_events ( SAMPLE_Events, (SET_BIT0 | SET_BIT1 | SET_BIT2), KAL_OR);
      ♦♦♦♦♦♦♦♦
}

void SAMPLE_LISR(void)
{
       ♦♦♦♦♦♦♦♦
         kal_activate_hisr(SAMPLE_Hisr);
       ♦♦♦♦♦♦♦♦
}

kal_status SAMPLE_consumer1 (void)
{
       kal_uint32     flags = 0;
       ♦♦♦♦♦♦♦♦
       /* Consumer will be resumed if and only if event flags SET_BIT0 and SET_BIT2 hold, having successfully
             retrieved the event flags, they will be cleared to zero. */
       kal_retrieve_eg_events(SAMPLE_Events, SET_BIT0 | SET_BIT2, KAL_AND_CONSUME, &flags, \
                                        KAL_SUSPEND);
       ♦♦♦♦♦♦♦♦
}

kal_status SAMPLE_consumer2 (void)
{
       kal_uint32     flags = 0;
       ♦♦♦♦♦♦♦♦
       /* Consumer will be resumed if and only if event flags SET_BIT1 is satisfied, having successfully retrieved the
          event flag, it will remain 1. */
       kal_retrieve_eg_events(SAMPLE_Events, SET_BIT1, KAL_OR, &flags, KAL_SUSPEND);
       ♦♦♦♦♦♦♦♦
}
```

8 Task Communication Management

Basically, MAUI is a message-passing system, any communication between two tasks or HISR and task are accomplished via message passing. The fundamental message packet is known as Inter Layer Message, abbreviated as ILM.

To prevent infinitive message delivery, each module is associated an ILM storage. Users must ensure he/she would not trigger message sending before the completeness of the previous request. Conventionally, user should allocate ILM in advanced, and put their data onto the storage; once delivered, the ILM storage will be released by the system.

Subsections following will give the detail description about data structure, data types and message passing flow.

8.1 Data Structures and Data Types

Tables below illustrate the data structure of an ILM in detailed.

local_para_struct ***(mcu\adaptation\include\app_ltlcom.h)***

Description:

Local parameter, acts as extended storage of an ILM, besides, single local parameter may be propagated to more than one destination. Therefore, header of a local parameter consists of two elements, ref_count tells the total number of consumers, and msg_len is used in tracking the storage length.

Data Type	Element	Description
kal_uint8	ref_count	Number of references, must not exceed 255.
kal_uint16	msg_len	Message length in terms of bytes.

peer_buff_struct ***(mcu\adaptation\include\app_ltlcom.h)***

Description:

Peer buffer, which also serves as additional storage while message passing. Similar with local parameter, single peer buffer may be propagated to more than one destination, therefore, ref_count tells the total number of consumers, and pdu_len is used in tracking the length of peer data (pdu). It is different from local parameter in the behavior of message accessing. Further description could be found in the next sub-section.

Data Type	Element	Description
kal_uint16	pdu_len	Message length in terms of bytes.
kal_uint8	ref_count	Number of references, must not exceed 255 times.
kal_uint8	pb_resvered	Padding element.
kal_uint16	free_header_space	Free space from head of a peer buffer.
kal_uint16	free_tail_space	Free space from tail of a peer buffer.

ilm_struct ***(mcu\adaptation\include\app_ltlcom.h)***

Description:

Fundamental unit of a message packet.

Data Type	Element	Description
module_type	src_mod_id	Source module ID.

module_type	dest_mod_id	Destination module ID.
sap_type	sap_id	SAP ID.
msg_type	msg_id	Message ID.
local_para_struct	*local_para_ptr	Pointer of local parameter.
peer_buff_struct	*peer_buff_ptr	Pointer of peer buffer.

kal_msgq_info ***(mcu\kal\include\kal_release.h)***

Description:

Data structure dedicated for queue information, including number of pending messages and total number of queue entries..

Data Type	Element	Description
kal_uint32	pending_msgs	Number of pending messages.
kal_uint32	max_msgs	Total number of queue entries.

kal_queue_stat_type, *kal_internal_queue_statistics ***(mcu\kal\nuclues\include\kal_nucleus.h)***

Description:

This data structure aims at tracking the statistical data of currently pending, as well as maximal number of pending messages. It is valid if and only if DEBUG_KAL, together with DEBUG_ITC are defined.

Data Type	Element	Description
kal_uint16	current_num_msgs	Tracking the number of pending messages.
kal_uint16	max_num_msgs_enqued	Tracking the maximal number of pending messages.

kal_queue_type, * kal_internal_msgqid ***(mcu\kal\nuclues\include\kal_nucleus.h)***

Description:

Defines the control block of message queue, slightly different if DEBUG_KAL and DEBUG_ITC are defined (light yellow background color).

Data Type	Element	Description
kal_os_queue_type	queue_id	Inherits from Nucleus Plus Queue control block.
Kal_uint16	max_msg_size	Maximal message size in terms of Bytes; which is now always equal to size of **ilm_struct**.
kal_internal_queue_statistics	q_stat	Pointer of queue statistical record.

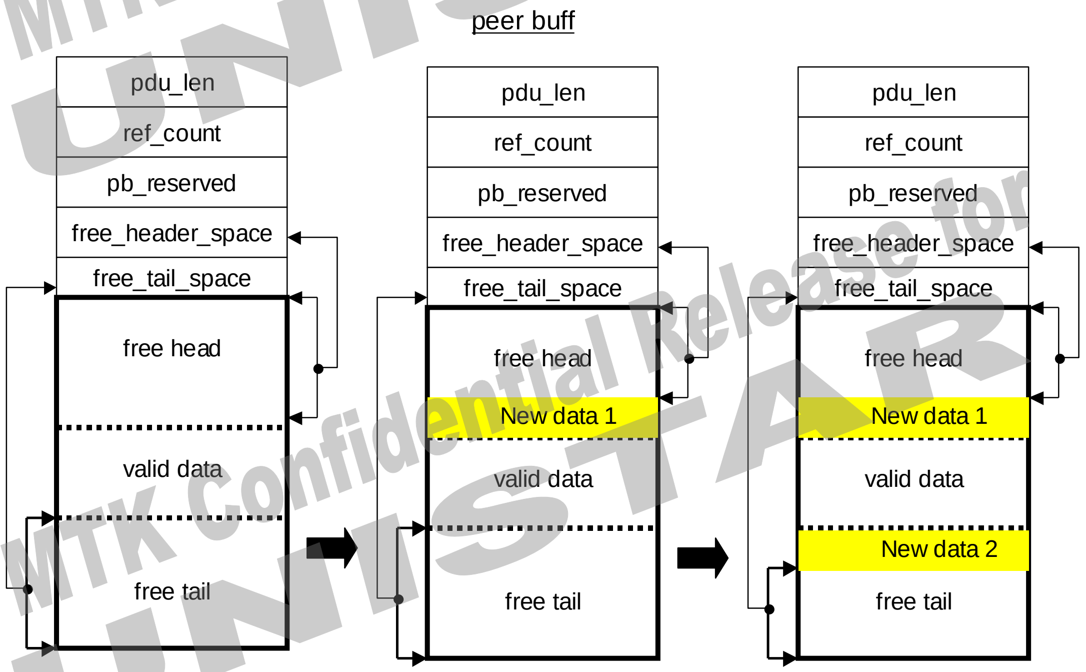
Data Types	Description
sap_type	Enumeration type of SAP ID.
msg_type	Enumeration type of message ID, must be leading with "MSG_ID_".
kal_msgqid	Identity of a message queue, which is internally equivalent to kal_internal_msgqid.

Global variable	Description
module_ilm_g	Array of **ilm_struct,** number of the entries is determined by enum of module ID plus 16 customer-defined tasks.

8.1.1 Access behavior of local parameter and peer buffer

A local parameter or peer buffer is normally created by provider, which is a task in charge of constructing buffer, and will be consumed by one or many consumers. Provider and/or consumer may call **hold_local_para( )** or **hold_peer_buff( )** to increment the variable before propagation; value decrementing ought to be done at consumer site in-paired by calling **free_local_para( )** or **free_peer_buff( )**.

For local parameter, data is fixed once constructed. But peer buffer offers two appending manners, either growing up from head (**prepend_to_peer_buff( )**)or down from tail (**append_to_peer_buff( )**). Figures below show the two appending schemes.



Data appending at front and tail of a peer buffer

8.2 Task Communication APIs

allocate_ilm

Prototype: ilm_struct * allocate_ilm (module_type module_id)
Header file: stack_ltlcom.h
Input: *rmodule_id* is source module ID, which is going to deliver an ILM.
Output: Pointer an ILM.
Description: ILM storage for any message delivery **must be allocated** from an array pool of type ilm_struct, namely **module_ilm_g** by sender. The array pool is indexed by sender's module ID. In case of violation, fatal error "Send an un-allocated ILM to external queue" (fatal error code 1 = 0x432, error code 2 = pointer of ILM) would be encountered. On the contrary, if sender attempts to send message before completeness of the previous delivery, fatal error again, "ILM is already allocated" (fatal error code 1 = 0x431, error code 2 = module ID).

append_to_peer_buff

Prototype: void append_to_peer_buff (peer_buff_struct *peer_buff_ptr, void *tail_data_ptr, kal_uint8 tail_len)
Header file: stack_ltlcom.h
Input: *peer_buff_ptr* is pointer of **peer_buff_struct**, from where new data pointed by *tail_data_ptr* will be appended from end of pdu, its length (*tail_len*) is given in units of Bytes.
Description: As shown in Figure 1, a data record could be appended from tail of peer buffer; after appending, system will update the following information,
peer_buff_ptr->free_tail_space = *peer_buff_ptr->free_tail_space* - *tail_len*;
peer_buff_ptr->pdu_len = *peer_buff_ptr->pdu_len* + *tail_len*;
Remark: Fatal error "append_to_peer_buff: insufficient tail memory" (fatal error code 1 = 0x10, error code 2 = 0x01) happens if either conditions holds,

1. *peer_buff_ptr* is NULL,
2. *tail_len* is zero,
3. *peer_buff_ptr->free_tail_space* is less than *tail_len*.

cancel_ilm

Prototype: kal_bool cancel_ilm (module_type module_id)
Header file: stack_ltlcom.h
Input: *rmodule_id* is source module ID, which is going to cancel its previous delivery.
Output: KAL_TRUE if source module has really allocated an ILM, and system has decremented reference count of both local parameter and peer buffer if existed, also, if reference count becomes zero, system will automatically release them. KAL_FALSE if source module not yet allocates an ILM.
Description: This function is used to halt the delivery of an ILM, system will free the previously allocated ILM. If either local parameter pointer or peer buffer pointer is not NIL, these buffers will be released as well.

construct_local_para

Prototype: void* construct_local_para(local_para_size, direction)
Header file: app_ltlcom.h
Input: *local_para_size* is size of local parameter in terms of bytes, *direction (TD_RESET)* is used to enforce zero-initialization of the local parameter.
Output: Void pointer.
Description: This is macro of

void* construct_int_local_para (kal_uint16 local_para_size, kal_char* file_ptr, kal_uint32 line)
if DEBUG_KAL is defined, otherwise, it is defined as
void* construct_int_local_para (kal_uint16 local_para_size).
In **construct_int_local_para( )**, system will get buffer according to size specified in *local_para_size*, if it is called by TST task, system allocates buffer from TST buffer, otherwise from control buffer. The return buffer pointer will then be casted as pointer of **local_para_struct**, the first byte of the buffer, or ref_count will be initialized to 1, and the second 2 bytes of the buffer, or msg_len would be filled as *local_para_size*.
From 06A w06.41, you can specify a TD_RESET (in the 2nd argument "direction") to let system do zero-initialization for you. For example:
construct_local_para(sizeof(my_struct), TD_RESET);
This will clear the buffer body to zero (note: not include the local parameter header).

Remark: Be very careful that, size of **local_para_struct** must be taken into account.

construct_peer_buff

Prototype: void *construct_peer_buff (pdu_len, header_len, tail_len, direction)
Header file: app_ltlcom.h
Input: *pdu_len* is size of peer buffer, *header_len* is size of header buffer, t*ail_len* is size of tail buffer, and *direction* is unused.
Output: Void pointer.
Description: This is macro of
void* construct_int_peer_buff(kal_uint16 pdu_len, kal_uint16 header_len, kal_uint16 tail_len, kal_char* file_name_ptr, kal_uint32 line)
if DEBUG_KAL is defined, otherwise, it is defined as
void* construct_int_peer_buff(kal_uint16 pdu_len, kal_uint16 header_len, kal_uint16 tail_len)).
Similar with **construct_int_local_para( )**, system is responsible to allocate buffer for peer buffer in **construct_int_peer_buff( )**, total buffer size is summation of sizeof(***peer_buff_ptr**), *header_len* , *pdu_len* and *tail_len*, if it is called by TST task, system allocates buffer from TST buffer, otherwise from control buffer. The return buffer pointer will then be casted as pointer of **local_para_struct**, the first byte of the buffer, or ref_count will be initialized to 1, and the second 2 bytes of the buffer, or msg_len would be filled as *local_para_size*.

free_ilm

Prototype: void free_ilm (ilm_struct* ilm_ptr)
Header file: stack_ltlcom.h
Input: *ilm_ptr* is pointer of ilm_struct, which is going to be freed.
Description: This is macro of
void free_int_ilm (ilm_struct *ilm_ptr, kal_char* file_name, kal_uint32 line)
if DEBUG_KAL is defined, otherwise, it is defined as
void free_int_ilm(ilm_struct *ilm_ptr).
If *ilm_ptr->peer_buff_ptr* is not NULL then decrement the associated reference count (*ilm_ptr->peer_buff_ptr->ref_count*), if it becomes zero, system will free the buffer directly, and resets the peer buffer pointer to NULL. The same algorithm will be applied on local parameter as well.

free_local_para

Prototype: void free_int_local_para (local_para_struct *local_para_ptr, kal_char* file, kal_uint32 line)
void free_int_local_para (local_para_struct *local_para_ptr)
Header file: stack_ltlcom.h

Input: *local_para_ptr* is pointer of **local_para_struct** to be released.
Description: If *local_para_ptr* is not NULL then decrement the associated reference count (*local_para_ptr->ref_count*), if it becomes zero, system will free the buffer directly! It is user's responsibility to reset the relative pointer kept in ilm_struct if needed.

Note: **There is another re-entrant function, free_int_local_para_r, which has protection on access to reference count. One who may have race condition problem of reference count could use the macro, free_local_para_r, to avoid such error. The usage is just the same as free_local_para.**

free_peer_buff

Prototype: void free_int_peer_buff(peer_buff_struct *pdu_ptr, kal_char* file, kal_uint32 line)
void free_int_peer_buff(peer_buff_struct *pdu_ptr)
Header file: stack_ltlcom.h
Input: *pdu_ptr* is pointer of **peer_buff_struct** to be released.
Description: If *pdu_ptr* is not NULL then decrement the associated reference count (*pdu_ptr->ref_count*), if it becomes zero, system will free the buffer directly! It is user's responsibility to reset the relative pointer kept in ilm_struct if needed.

Note: **There is another re-entrant function, free_int_peer_buff_r, which has protection on access to reference count. One who may have race condition problem of reference count could use the macro, free_peer_buff_r, to avoid such error. The usage is just the same as free_peer_buff.**

hold_local_para

Prototype: kal_bool hold_local_para (local_para_struct *local_para_ptr)
Header file: stack_ltlcom.h
Input: *local_para_ptr* is pointer of **local_para_struct**, whose content is going to be reserved for some consumer.
Output: KAL_FALSE if *local_para_ptr* is NULL pointer, otherwise, KAL_TRUE.
Description: If *local_para_ptr* is not NULL, increment its reference counter (*local_para_ptr -> ref_count*) by 1.

Note: **There is another re-entrant function, hold_local_para_r, which has protection on access to reference count. One who may have race condition problem of reference count could use this API to avoid such error. The usage is just the same as hold_local_para.**

get_local_para_ptr

Prototype: void * get_local_para_ptr (local_para_struct *local_para_ptr, kal_uint16 *local_para_len_ptr)
Header file: stack_ltlcom.h
Input: *local_para_ptr* is pointer of **local_para_struct**, and *local_para_len_ptr* will be used to return message length.
Output: Pointer, which targets at the starting address of local parameter buffer.
Description: This function call is useful in retrieving length of local parameter buffer and its start address.

get_pdu_ptr

Prototype: void * get_pdu_ptr (peer_buff_struct *peer_buff_ptr, kal_uint16 *length_ptr)
Header file: stack_ltlcom.h
Input: *peer_buff_ptr* is pointer of **peer_buff_struct**, and *length_ptr* will be used to return length of current pdu.
Output: Pointer, which targets at the starting address of peer data.

Description: This function call is useful in retrieving length of peer buffer and start address of peer data, the later is obtained from ((kal_uint8 *)peer_buff_ptr + sizeof(*peer_buff_ptr) + peer_buff_ptr->free_header_space).

hold_peer_buff

Prototype: kal_bool hold_peer_buff (peer_buff_struct *peer_buff_ptr)
Header file: stack_ltlcom.h
Input: *peer_buff_ptr* is pointer of **peer_buff_struct**, whose content is going to be reserved for some consumer.
Output: KAL_FALSE if *peer_buff_ptr* is NULL pointer, otherwise, KAL_TRUE.
Description: If *peer_buff_ptr* is not NULL, increment its reference counter (*peer_buff_ptr-> ref_count*) by 1.

Note: **There is another re-entrant function, hold_peer_buff_r, which has protection on access to reference count. One who may have race condition problem of reference count could use this API to avoid such error. The usage is just the same as hold_peer_buff.**

msg_get_ext_queue_info

Prototype: kal_bool msg_get_ext_queue_info (kal_msgqid task_ext_qid, kal_uint32 *messages)
Header file: stack_ltlcom.h
Input: *task_ext_qid* is pointer of queue identity, the returned information would be returned via *messages*.
Output: KAL_TRUE if information retrieval successfully done, otherwise KAL_FALSE.
Description: This is an exported function for querying number of pending messages.

msg_get_ext_queue_length

Prototype: kal_bool msg_get_ext_queue_length (kal_msgqid task_ext_qid, kal_uint32 *length)
Header file: stack_ltlcom.h
Input: *task_ext_qid* is pointer of queue identity, the returned information would be returned via *length*.
Output: KAL_TRUE if information retrieval successfully done, otherwise KAL_FALSE.
Description: This is an exported function for querying total number of queue entries.

msg_send_ext_queue

Prototype: kal_bool msg_send_ext_queue (ilm_struct *ilm_ptr)
Header file: stack_ltlcom.h
Input: *ilm_ptr* is pointer of ILM, which is going to be delivered.
Output: KAL_TRUE if message is delivered successfully, otherwise, return KAL_FALSE.
Description: This is an exported function for message delivery; destination is always **external** queue of task specified in ilm_ptr-> dest_mod_id. Destination task's external queue is targeted through the statement below,
task_info_g [**mod_task_g** [ilm_ptr->dest_mod_id]].task_ext_qid
Soon after delivery, the ILM structure, which is previously get via **allocate_ilm( )** will be returned to the system.

msg_send_ext_queue_to_head

Prototype: kal_bool msg_send_ext_queue_to_head(ilm_struct *ilm_ptr)
Header file: stack_ltlcom.h
Input: *ilm_ptr* is pointer of ILM, which is going to be delivered.
Output: KAL_TRUE if message is delivered successfully, otherwise, return KAL_FALSE.
Description: This is an exported function for message delivery; destination is always **external** queue of task specified in ilm_ptr-> dest_mod_id. Destination task's external queue is targeted through the statement below,
task_info_g [**mod_task_g** [ilm_ptr->dest_mod_id]].task_ext_qid

Soon after delivery, the ILM structure, which is previously get via **allocate_ilm( )** will be returned to the system.
It is different from msg_send_ext_queue in manner of message delivery behavior. API msg_send_ext_queue always sends a message to end of queue, forming the last-in-last-out behavior. While msg_send_ext_queue_to_head always sends a message to front of queue, forming the last-in-first-out behavior.
Users may take such feature to force the early process of a high priority message.

Remark: Available from W05.32.

msg_send_int_queue

Prototype: kal_bool msg_send_int_queue (ilm_struct *ilm_ptr)

Header file: stack_ltlcom.h

Input: *ilm_ptr* is pointer of ILM, which is going to be delivered.

Output: KAL_TRUE if message is delivered successfully, otherwise, return KAL_FALSE.

Description: This is an exported function for message delivery; destination is always **internal** queue of task specified in ilm_ptr-> dest_mod_id. If the destination task has no internal queue, fatal error would be trapped. Soon after delivery, the ILM structure, which is previously get via **allocate_ilm( )** will be returned to the system.

prepend_to_peer_buff

Prototype: void prepend_to_peer_buff (peer_buff_struct *peer_buff_ptr, void *header_data_ptr, kal_uint8 header_len)

Header file: stack_ltlcom.h

Input: *peer_buff_ptr* is pointer of **peer_buff_struct**, from where new data pointed by *header_data_ptr* will be appended at front of pdu, its length (*header_len*) is given in units of Bytes.

Description: As shown in Figure 1, a data record could be appended at head of peer buffer; after appending, system will update the following information,
peer_buff_ptr-> free_header_space = *peer_buff_ptr-> free_header_space* - *header_len*;
peer_buff_ptr->pdu_len = *peer_buff_ptr->pdu_len* + *header_len*;

Remark: Fatal error “prepend_to_peer_buff : insufficient header memory” (fatal error code 1 = 0x10, error code 2 = 0x02) happens if either conditions holds,

4. *peer_buff_ptr* is NULL,
5. *header_len* is zero,
6. *peer_buff_ptr->free_header_space* is less than *header_len*.

receive_msg_ext_q

Prototype: kal_status receive_msg_ext_q (kal_msgqid task_ext_qid, ilm_struct *ilm_ptr)

Header file: stack_ltlcom.h

Input: *task_ext_qid* is ID of destination queue, from which a message will be retrieved, data packet will then be written to *ilm_ptr*.

Output: KAL_SUCCESS if message is retrieval is successfully done, otherwise fatal error with error code 1 = 0x308 would be trapped.

Description: This is the unified entrance for external queue message retrieval, if the destination queue is empty, caller would suspend it-self endlessly.

Remark: Unlike **msg_send_ext_queue( )**, the ILM pointer *ilm_ptr* is a privately maintained pointer, which is not requested from the system.

receive_msg_int_q

Prototype:	kal_bool receive_msg_int_q (task_indx_type task_indx, ilm_struct *ilm_ptr)
Header file:	stack_ltlcom.h
Input:	*task_indx* is task ID, while *ilm_ptr* acts as storage.
Output:	KAL_FALSE if internal queue is empty, otherwise, KAL_TRUE.
Description:	Retrieving an ILM from task internal queue, if there is an available ILM, KAL_TRUE will be returned, and the content is kept at *ilm_ptr.*
Remark:	Unlike **msg_send_int_queue( )**, the ILM pointer *ilm_ptr* is a privately maintained pointer, which is not requested from the system.

remove_hdr_of_peer_buff

Prototype:	void remove_hdr_of_peer_buff (peer_buff_struct *peer_buff_ptr, kal_uint8 hdr_len)
Header file:	stack_ltlcom.h
Input:	*peeer_buff_ptr* is destination peer buffer whose peer data is going to be removed from header by *hdr_len* Bytes.
Description:	Contrary to **prepend_to_peer_buff( )**, this function aims to remove peer data from header of peer buffer, total number of data to be removed is expressed in *hdr_len*. After removing, system updates the peer buffer structure as below, *peer_buff_ptr->free_header_space = peer_buff_ptr->free_header_space + hdr_len*; *peer_buff_ptr->pdu_len = peer_buff_ptr->pdu_len - hdr_len*;

remove_tail_of_peer_buff

Prototype:	void remove_tail_of_peer_buff (peer_buff_struct *peer_buff_ptr, kal_uint8 tail_len)
Header file:	stack_ltlcom.h
Input:	*peeer_buff_ptr* is destination peer buffer whose peer data is going to be removed from tail by *tail_len* Bytes.
Description:	Contrary to **append_to_peer_buff( )**, this function aims to remove peer data from tail of peer buffer, total number of data to be removed is expressed in *tail_len.* After removing, system updates the peer buffer structure as below, *peer_buff_ptr-> free_tail_space = peer_buff_ptr-> free_tail_space + tail_len*; *peer_buff_ptr->pdu_len = peer_buff_ptr->pdu_len - tail_len*;

update_peer_buff_hdr

Prototype:	void update_peer_buff_hdr (peer_buff_struct *peer_buff_ptr, kal_uint8 new_hdr_len, kal_uint16 new_pdu_len, kal_uint8 new_tail_len)
Header file:	stack_ltlcom.h
Input:	*peer_buff_ptr* is pointer of **peer_buff_struct**, its pdu length, free space of buffer header and tail are going to be replaced with *new_pdu_len* , *new_hdr_len* and *new_tail_len* respectively.
Description:	Reinitialize peer buffer length, including header and tail free space, together with pdu length.

8.3 Example

A typical message passing flow is illustrated as following,

(A) Sender allocates, packs and delivers an ILM

```
typedef struct {
    kal_uint8 ref_count;
    kal_uint16 msg_len;
    kal_uint8 access_id;
    kal_uint8 file_idx;
    kal_uint16 para;
} sender_buffer_struct;

void sender(void)
{
       ilm_struct   *sender_ilm = allocate_ilm(MOD_SENDER);    /* Necessary */
       sender_buffer_struct * local_data;

       local_data = (sender_buffer_struct *) construct_local_para ( sizeof(sender_buffer_struct), TD_CTRL );

        local_data->access_id = 0;
        local_data->file_idx = SENDER_ID_FILE;
        local_data->para = 1;

       sender_ilm.src_mod_id = MOD_SENDER;
       sender_ilm.dest_mod_id = MOD_RECEIVER;
       sender_ilm.msg_id = MSG_ID_SENDER_TO_RECEIVER;
       sender_ilm.sap_id = NULL;
       sender_ilm.local_para_ptr = (local_para_struct*) local_data;
       sender_ilm.peer_buff_ptr = 0;

       msg_send_ext_queue(sender_ilm);
}
```

(B) Receiver processes an ILM

There are two alternative approaches in processing local parameter or peer buffer, **free_local_para( )** is called artificially in approach 1, whilst local parameter is totally in-charged by system in approach 2. Note that, if **free_local_para( )** is called, be ensured that, local parameter point is reset to NULL by the last consumer, otherwise, in **free_ilm( )**, it may get the pointer which might be owned by others, and finally, mistakenly updates others buffer! Programmers could transfer the ILM manipulation to system by simply calling **free_ilm( )**, where system will reset the local parameter pointer and peer buffer pointer to NULL automatically if they are freed.

Approach 1, Manually maintain local parameter pointer.

```
void receiver(void)
{
    ilm_struct current_ilm;

    while (1)
    {
        receive_msg_ext_q( task_info_g [ INDX_RECEIVER ].task_ext_qid, &current_ilm);
        switch (current_ilm. msg_id)
        {
            case MSG_ID_SENDER_TO_RECEIVER :
                        ♦♦♦♦♦♦♦♦
                        free_local_para(current_ilm. local_para_ptr);
                        current_ilm. local_para_ptr = NULL;   /* Necessary if it is the last consumer */
        }
        free_ilm(&current_ilm);
    }
}
```

Approach 2, System is responsible for maintaining local parameter pointer.

```
void receiver(void)
{
    ilm_struct current_ilm;

    while (1)
    {
        receive_msg_ext_q( task_info_g [ INDX_RECEIVER ].task_ext_qid, &current_ilm);
        switch (current_ilm. msg_id)
        {
            case MSG_ID_SENDER_TO_RECEIVER :
                        ♦♦♦♦♦♦♦♦
        }
        free_ilm(&current_ilm);
    }
}
```

9 Timer Management

There are 3 types of timers provided on MAUI, they are KAL timer, stack timer and scheduled event respectively. Stack timer and scheduled event are for protocol stack, also upper layer applications; KAL timer is a wrapper of Nucleus Plus timer component, it the most precise timer, its call-back function is done at HISR level, therefore, it is restricted for OS and driver layer only. Any application violates the restriction may suffer from serous failure on mobility features.

On MAUI, the minimal time tick is in terms of TDMA timer, 4.615ms each unit.

9.1 Descriptions

9.1.1 KAL timer

Basically, KAL timer is a wrapper of Nucleus Plus Timer Management. It offers the most accurate timer period, because the timeout handler is done by Timer HISR. Inheriting from the design, any callback function registered on KAL timer must take the processing latency into account; the longer processing period delays the subsequent processes more.

A statistical data structure is integrated on KAL timer control block for the tracking of following items,

- Number of timer cancellations,
- Number of timer expirations,
- Timer status, either KAL_TIMER_CREATED, KAL_TIMER_SET, KAL_TIMER_CANCELED or KAL_TIMER_EXPIRED

Note that, timer statistic is available if DEBUG_KAL and DEBUG_TIMER are defined.

9.1.2 Stack timer

Stack timer is the second accurate timer provided on the system. It is designed to relieve system loading on processing KAL timer timeout handler, also the concurrent programming issues. Stack timer is different from KAL timer in the behavior of handling timeout procedure. The later is achieved by registering callback function on the dedicated KAL timer, while the former is done by message delivery on stack timer timeout routine. The timeout unit processes the correspondence procedure on its task context. Obviously, the accuracy of timeout notification is sacrificed on the flow of message delivery, besides, system scheduling status and scheduling overhead.

Each stack timer associates a KAL timer, and it is more suitable for down-counting timer rather than event that frequently happens, in addition, late timeout notification is allowed.

The typical ILM content of a stack timer timeout message as below,

Element	Value	Description
src_mod_id	MOD_TIMEr	The source module ID is definitely MOD_TIMER.
dest_mod_id	stack_timer->dest_mod_id	The destination module ID is specified on stack timer control block.
msg_id	MSG_ID_TIMER_EXPIRY	It is the constant mesasge ID.
sap_id	STACK_TIMER_SAP	It is the constant SAP ID.
local_para_ptr	stack_timer	Packing the stack timer control block as local parameter. **It**

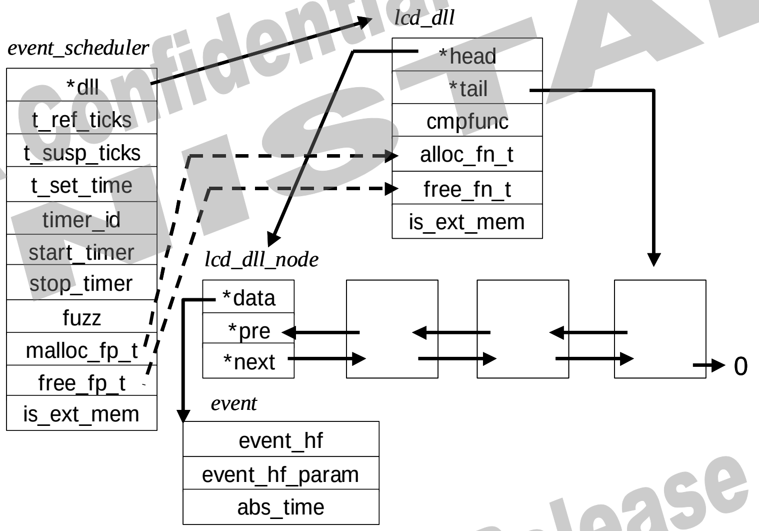
		is not allowed to hold the local parameter of a stack timer timeout message, because there is no local parameter header in the stack timer structure.
peer_buff_ptr	NULL	Peer buffer is always excluded from the stack timer timeout message.

On expiration, stack timer timeout handler will be waken-up by the timer HISR, it posts the expiry ILM to the destination task via timeout ILM; finally, the task being notified will be scheduled by the OS, and start to process the expiry ILM.

However, if the task stops the timer before receiving expiry ILM, the task shouldn't process the expiry ILM anymore. To cover the circumstance, two function calls are provided to determine if the expiry message should be taken, they are **stack_is_time_out_valid( )** and **stack_process_time_out( )**. Note that, they must be called in-paired. Please refer to their APIs and example on section 6.4.1 for more detailed.

9.1.3 Event scheduler

The scheduled event or event scheduler another alternative timer special for frequently happen events, which needs non-accurate timeout period. It is especially suitable for upper layer applications, which just need the timeout notification with less accuracy, for instances, a down-counting counter used on tracking the backlight on/off, termination of an audio playing etc.



Internal data structure of an event scheduler

Figure above depicts the data structure of an event scheduler, please refer to section 6.2.3 for more detailed. Usually, **stack_start_timer( )**, **stack_stop_timer( )**, **stack_is_time_out_valid( )** and **stack_process_time_out( )** act as the based timer for an event scheduler. **Users who are using an event scheduler must know the exact event ID very well.** An event ID is actually a pointer, from where system will allocate and release pointer while setting an event. Memory de-allocation will be done in **evshed_cancel_event( )** and **evshed_timer_handler( )**. In **kal_cancel_event( )**, system releases memory by referencing the event ID, henceforth, users must ensure it is reset at the registered event timeout handler. Otherwise, users may accidentally cancel the event set by the new owner, who set an event soon after the previous was timeout.

Function call **evshed_timer_handler( )** is dedicated for processing a series of timeout event registered on an event scheduler.

9.2 Data Structures and Data Types

9.2.1 KAL timer

Tables below illustrate the data structure and data types used by KAL timer.

kal_timer_stat_type, *kal_internal_timer_statistics *(mcu\kal\nucleus\include\kal_nucleush)*		
Description: KAL internal structure for tracking the statistical data of a KAL timer, valid if both DEBUG_KAL and DEBUG_TIMER are defined.		
Data Type	**Element**	**Description**
kal_timer_state	timer_state	Timer status.
kal_char*	timer_name	Character string, which tells the name of a timer.
kal_uint16	num_times_expired	Number of expirations.
kal_uint16	num_times_canceled	Number of cancellations.

kal_timer_type, *kal_internal_timerid *(mcu\kal\nucleus\include\kal_nucleush)*		
Description: KAL internal structure for maintaining KAL timer, among the elements, timer_stat is valid if DEBUG_KAL and DEBUG_TIMER are defined.		
Data Type	**Element**	**Description**
kal_os_timer_type	timer_id	A pointer to Nucleus Plus timer.
kal_timer_func_ptr	func_ptr	Timeout handler routine.
void *	timer_param_ptr	Timer parameters.
kal_uint32	set_time	Time ticks being set.
kal_internal_timer_statistics	timer_stat	Timer statistical pointer.

Data Types	Description
kal_timer_state	Tracking the KAL timer status, either KAL_TIMER_CREATED, KAL_TIMER_SET, KAL_TIMER_CANCELED or KAL_TIMER_EXPIRED.
kal_timerid	Timer ID of KAL timer, which is internally equivalent to kal_internal_timerid.

9.2.2 Stack timer

Tables below illustrate the data structure and data types used by stack timer.

stack_timer_struct *(mcu\stacklib\include\stack_timer.h)*		
Description: Internal data structure for stack timer management.		
Data Type	**Element**	**Description**
module_type	dest_mod_id	Destination module ID.
kal_timerid	kal_timer_id	Identify of a KAL timer.
kal_uint16	timer_indx	Timer index, which is maintained by the users.
stack_timer_status_type	timer_status	Status of stack timer.
kal_uint8	invalid_time_out_count	Number of cancellation before the processing of

		expiry message. **It is used in the functions stack_process_time_out and stack_stop_timer. Both of these functions are supposed to be called by the modules of the same task for the same stack timer. There will be race conditions in case of concurrent access from multiple tasks..**

Data Types	Description
stack_timer_status_type	It is an enumeration type of stack timer status, STACK_TIMER_INITIALIZED, STACK_TIMER_NOT_RUNNING = STACK_TIMER_INITIALIZED, STACK_TIMER_RUNNING, STACK_TIMER_NOT_TIMED_OUT = STACK_TIMER_RUNNING, STACK_TIMER_EXPIRED, STACK_TIMER_TIMED_OUT = STACK_TIMER_EXPIRED, STACK_TIMER_STOPPED

9.2.3 Event scheduler

Tables below illustrate the data structure and data types used by an event scheduler.

lcd_dll_node ***(mcu\stacklib\include\lcd_dll.h)***

Description:
Fundamental unit of double linked list, abbreviated as lcd_dll.

Data Type	Element	Description
void *	data	Pointer of actual storage.
lcd_dll_node *	prev	The previous pointer.
lcd_dll_node *	next	The next pointer.

lcd_dll ***(mcu\stacklib\include\lcd_dll.h)***

Description:
Internal data structure for double-linked list will allocation and de-allocation handler.

Data Type	Element	Description
lcd_dll_node *	head	Header of double-linked list, which points to the first lcd_dll_node.
lcd_dll_node *	tail	Tail of double- liked list, which points to the last lcd_dll_node.
lcd_cmpfunc	cmpfunc	Compare function used for inserting a node..
malloc_fp_t	alloc_fn_p	Allocation function used for allocating a lcd_dll_node.
free_fp_t	free_fn_p	De-allocation function used for de-allocating a lcd_dll_node.

event_scheduler ***(mcu\stacklib\include\event_sched.h)***

Description:
Internal data structure for the management of event scheduler.

Data Type	Element	Description
lcd_dll *	dll	Event's double-linked list.
lcd_dll_node *	expired_dllhead	Event's expired dll list split in **evshed_timer_handler( )**.
kal_uint32	t_ref_ticks	Scheduler's reference time base.
kal_uint32	t_susp_ticks	Scheduler's reference suspend time base for suspend operation.
kal_bool	is_suspend	Scheduler's suspension flag.
kal_uint32	t_set_time	Scheduler's timer's set time.
void *	timer_id	Timer ID.
void	(*start_timer)(void *, unsigned int)	Start timer handler.
void	(*stop_timer)(void *)	Stop timer handler.
malloc_fp_t	alloc_fn_p	Event's storage for allocation handler.
malloc_fp_t	free_fn_p	Event's storage for de-allocation handler.

Data Types	Description
typedef void * (*malloc_fp_t)(unsigned int)	Function type used as memory allocation.
typedef void (*free_fp_t)(void *)	Function type used as memory de-allocation.
typedef int (*lcd_cmpfunc)(const void *, const void *)	Comparison function embedded on double-linked list.
typedef lcd_dll_node *eventid	An event node on event scheduler, it will be known an event ID subsequently.

9.3 APIs

9.3.1 KAL timer

kal_cancel_timer

Prototype: void kal_cancel_timer (kal_timerid ext_timer)
Header file: kal_release.h
Input: *ext_timer* is destination KAL timer to be cancelled.
Description: This service is used to cancel a timer; in DEBUG_TIMER mode, it is switches to KAL_TIMER_CANCELED if it is successfully cancelled.

kal_create_timer

Prototype: kal_timerid kal_create_timer (kal_char* timer_name_ptr)
Header file: kal_release.h
Input: *timer_name_ptr* is name of the timer to be created.
Output: A KAL timer ID will be returned it is successfully created.
Description: This service is used for creating KAL timer; in DEBUG_TIMER, soon after creation, the timer stays at KAL_TIMER_CREATED state. **It is strongly suggested to call this function only at system initialization stage, and the related data allocated for the timer could not be freed once it's created.**

kal_get_time

Prototype:	void kal_get_time (kal_uint32* ticks_ptr)
Header file:	kal_release.h
Input:	*ticks_ptr* is used for returning current time ticks.
Description:	Users could use this function to query current time ticks.

kal_get_time_remaining

Prototype:	kal_uint32 kal_get_time_remaining (kal_timerid ext_timer_id)
Header file:	kal_release.h
Input:	*ext_timer_id* is identity of a KAL timer being queried.
Output:	Remaining timeout period expressed in 32bits unsigned integer.
Description:	This is used in retrieving the remaining timeout period, also in units of time ticks.

kal_get_timer_statistics

Prototype:	void kal_get_timer_statistics (kal_timerid ext_timer_id, kal_timer_statistics* ext_timer_stat)
Header file:	kal_release.h
Input:	*ext_timer_id* is ID of KAL timer being queried, and *ext_timer_stat* is used for returning the statistical record.
Description:	Users could use this function for retrieving the current statistical record.

kal_set_timer

Prototype:	void kal_set_timer (kal_timerid ext_t_id, kal_timer_func_ptr handler_func_ptr, void* handler_param_ptr, kal_uint32 delay, kal_uint32 reshedule_time)
Header file:	kal_release.h
Input:	*ext_t_id* is identity of a KAL timer, *handler_func_ptr* is pointer of timeout handler, *handler_param_ptr* is pointer to timeout handler's parameter, *delay* is timeout duration in terms of time ticks, the maximum value is 0xFFFFFFFF. Argument *reshedule_time* is auto rescheduled time.
Description:	This function call offers the service of setting a timeout period to the given KAL timer. In DEBUG_TIMER mode, a timer switches to KAL_TIMER_SET once it is set.

9.3.2 Stack timer

stack_init_timer

Prototype:	void stack_init_timer (stack_timer_struct *stack_timer, kal_char *timer_name, module_type mod_id)
Header file:	stack_timer.h
Input:	*stack_timer_struct* is the stack timer control block provided by the users, *timer_name* is the name of the stack timer, and *mod_id* is the module ID of the owner, to which an expiry ILM will be delivered.
Description:	This function provides the facility of initializing a stack timer; it is the users' responsibility to provide stack timer control block. Soon after creation, its status remains at STACK_TIMER_INITIALIZED or STACK_TIMER_NOT_RUNNING. **It is strongly suggested to call this function only at system initialization stage, and the related data allocated for the stack timer could not be freed once it's created.**

stack_is_time_out_valid

Prototype:	kal_bool stack_is_time_out_valid (stack_timer_struct *stack_timer)
Header file:	stack_timer.h
Input:	*stack_timer* is the destination stack timer being queried.
Output:	KAL_FALSE if the KAL timer id associated is NIL, or the stack timer has ever been stopped or cancelled before expiration; otherwise, KAL_TRUE.
Description:	This function is used for tracking the validity of a timeout notification, if it returns KAL_FALSE, the timeout handler should not be handled.

stack_process_time_out

Prototype:	void stack_process_time_out(stack_timer_struct *stack_timer)
Header file:	stack_timer.h
Input:	*stack_timer* is the destination stack timer to be processed.
Description:	Together with **stack_is_time_out_valid( )**, they are the paired functions used for identify validity of an expiry message. If an invalid expired message is received, for instance, the timer has been stopped, **stack_process_time_out( )** should be called to correctly maintain the control element invalid_time_out_count. It is in charge of decrementing invalid_time_out_count by 1, besides, switch the stack timer status to STACK_TIMER_NOT_RUNNING if it is in STACK_TIMER_EXPIRED.

stack_start_timer

Prototype:	void stack_start_timer (stack_timer_struct *stack_timer, kal_uint16 timer_indx, kal_uint32 init_time)
Header file:	stack_timer.h
Input:	*stack_timer_struct* is the stack timer to be set, *timer_indx* is the index of the timer, and *init_time* tells the timeout period, similar to KAL timer, the maximum value is also 0xFFFFFFFF.
Description:	This facility is used to start a stack timer; if the timeout period *init_time* is zero while setting, the expiry ILM will be sent to the destination module immediately, and system changes the stack timer status to STACK_TIMER_EXPIRED also. Otherwise, a stack timer is started, and the new status is STACK_TIMER_RUNNING or STACK_TIMER_NOT_TIMED_OUT. **It is allowed to be called when the stack timer is in running or time out state. If it's in running state, the new time out value would be applied if this function is called. On the other hand, if it's in time out state, the original time out message will still be processed, and a new timer will be scheduled.**

stack_stop_timer

Prototype:	stack_timer_status_type stack_stop_timer (stack_timer_struct *stack_timer)
Header file:	stack_timer.h
Input:	*stack_timer* is the destination stack timer to be stopped.
Output:	Current status of the stack timer.
Description:	On the termination of a stack timer, the processing procedure is highly dependent on it current status. If the stack timer stays at STACK_TIMER_INITIALIZED or STACK_TIMER_NOT_RUNNING, system does nothing except refilling its status to STACK_TIMER_NOT_RUNNING. If it is in STACK_TIMER_RUNNING at the mean time, system will cancel it directly. Having completely cancelled the stack timer, its status will be re-examined again; if it is ever switched to STACK_TIMER_EXPIRED during the period, the tracking element *invalid_time_out_count* will be incremented, besides, further changing the status to STACK_TIMER_STOPPED, however, the return will be STACK_TIMER_TIMED_OUT. Otherwise, the stack timer status and the return value will be STACK_TIMER_STOPPED consistently.

stack_timer_status

Prototype: stack_timer_status_type stack_timer_status (stack_timer_struct *stack_timer, kal_uint32 *time_remaining)

Header file: stack_timer.h

Input: *stack_timer* is the destination stack timer being queried, and *time_remaining* is used for returning the remaining timeout period.

Output: Current status of the stack timer.

Description: This function is used in retrieving the most updated status of the given stack timer, if the timer is being set and not yet timeout, system returns STACK_TIMER_NOT_TIMED_OUT, otherwise, STACK_TIMER_TIMED_OUT.

9.3.3 Event scheduler

evshed_cancel_event

Prototype: kal_int32 evshed_cancel_event (event_scheduler *es, eventid *eid)

Header file: event_sched.h

Input: *es* is pointer of an event scheduler, *eid* is the pointer to event ID.

Output: Time difference between current time and event scheduled time, in units of time ticks.

Description: This service is called for canceling an event, **system would reset the event ID (eid) to NULL before returning; however, caller should pay attention to its privately saved event id.**

evshed_delete_all_events

Prototype: void evshed_delete_all_events (event_scheduler *es)

Header file: event_sched.h

Input: *es* is pointer of an event scheduler.

Description: This function aims for removing all events from the double-linked list of an event scheduler.

evshed_get_rem_time

Prototype: kal_uint32 evshed_get_rem_time (event_scheduler *es, eventid eid)

Header file: event_sched.h

Input: *es* is pointer of an event scheduler, and *es* is pointer of event ID.

Output: The remaining time ticks.

Description: This function used to return the remaining time ticks of the dedicated event ID.

evshed_resume_all_events

Prototype: void evshed_resume_all_events (event_scheduler *es)

Header file: event_sched.h

Input: *es* is pointer of an event scheduler.

Description: This function aims to resume all of the events registered on an event scheduler.

evshed_set_event

Prototype: eventid evshed_set_event (event_scheduler *es, kal_timer_func_ptr event_hf, void *event_hf_param, kal_uint32 elapse_time)

Header file: event_sched.h

Input: *es* is pointer of an event scheduler, *event_hf* is the event timeout handler, *event_hf_param* points to event timeout handler's parameter and *elapse_time* tells the total elapse time in ticks.
Output: Pinter to the event ID.
Description: This service is provided for setting an event to an event scheduler; **system would allocate memory for event id, and return to the caller. If caller need to save such event id, please remember to reset it when cancel the event or on its expiration, because it is no longer valid.**

evshed_suspend_all_events

Prototype: void evshed_suspend_all_events (event_scheduler *es)
Header file: event_sched.h
Input: *es* is pointer of an event scheduler.
Description: This function aims to suspend all of the events registered on an event scheduler.

evshed_timer_handler

Prototype: void evshed_timer_handler (event_scheduler *es)
Header file: event_sched.h
Input: *es* is pointer of an event scheduler.
Description: This is the unified entrance of the timeout handler of the given event scheduler.

new_evshed

Prototype: event_scheduler *new_evshed (void *timer_id, void (*start_timer)(void *, unsigned int),
void (*stop_timer)(void *), kal_uint32 fuzz, malloc_fp_t alloc_fn_p,
free_fp_t free_fn_p, kal_uint8 max_delay_ticks)
Header file: event_sched.h
Input: *timer_id* is the base timer pointer of an event scheduler's base timer pointer, *start_timer* is the base timer start handler, *stop_timer* acts as base timer stop handler, *fuzz* is the event scheduler's event correction time offset, *alloc_fn_p* is the event allocation handler, *free_fn_p* is the event free handler and *max_delay_ticks* tells if it is an aligned timer.
Output: Pinter to the created event scheduler.
Description: This function is called for creating an event scheduler, users are responsible to provide timer start and stop handler, memory allocation and de-allocation function handlers. To optimize power saving, before system getting into sleep mode, it will find the first un-aligned timer, and system must wake up its timeout. An un-aligned timer is a timer whose timeout notification must not be delayed due to system sleep. On the contrary, an aligned timer allows the late timeout notification. The parameter *max_delay_ticks* tells if it is an aligned timer, it is extremely important for power saving issue. When setting max_delay_ticks to zero, the timer will be marked as an un-aligned timer. The timer will be marked as an aligned timer when the value of max_delay_ticks is 255.

From 05C W05.13, specifying other values (i.e., 1 ~ 254) will also mark the timer as an aligned timer. But the timeout notification won't be delayed more than max_delay_ticks. For example, you set max_delay_ticks to 10. The timeout notification of the timer maybe delayed due to the sleep mode. But the maximal delay will not exceed 10 ticks.

It is strongly suggested to call this function only in system initialization time, and the related data allocated for the event scheduler could not be freed once it's created.

9.4 Examples

Here is an integrated example of event scheduler and stack timer.

```
void ssdbg2_main (task_entry_struct * task_entry_ptr)
{
    kal_uint32 Task_Time = 0;
    char display_str[128];
    ilm_struct current_ilm;
    ssdbg1_localpara_struct *paraptr;
    ssdbg1_peerbuff_struct *peerbufptr;
    kal_uint8 *pduptr;
    kal_uint16 pdulength, i;

    /* initialize a stack timer as the base timer */
    stack_init_timer(&ssdbg2_context.ssdbg2_base_timer, "SSDBG2 Base Timer", MOD_SSDBG2);

    /* create scheduled event */
    ssdbg2_context.ssdbg2_event_scheduler_ptr = new_evshed(&ssdbg2_context.ssdbg2_base_timer,
                                           ssdbg2_start_base_timer, ssdbg2_stop_base_timer,
                                           0 , kal_evshed_get_mem, kal_evshed_free_mem, 0);

    /* set event */
    ssdbg2_context.ssdbg2_event_id = evshed_set_event(ssdbg2_context.ssdbg2_event_scheduler_ptr, \
                                    (kal_timer_func_ptr)ssdbg2_base_timer_timeout_hdler, NULL, 200);

    while (1) {
        Task_Time++;

        while (receive_msg_int_q(task_entry_ptr->task_indx, &current_ilm))
        {
            switch (current_ilm.msg_id)
            {
            case MSG_ID_TIMER_EXPIRY:
                /* check if the base timer is stopped or not */
                if (stack_is_time_out_valid(&ssdbg2_context.ssdbg2_base_timer)) {
                  /* invoke event's timeout handler */
                  evshed_timer_handler(ssdbg2_context.ssdbg2_event_scheduler_ptr);
                }
                stack_process_time_out(&ssdbg2_context.ssdbg2_base_timer);
                break;
            default:
                break;
            }
            free_ilm(&current_ilm);  // Inside the function, it does nothing for a timer message
        }
    }
}
```

```
void ssdbg2_start_base_timer(void *base_timer_ptr,unsigned int time_out)
{
      stack_start_timer((stack_timer_struct *)base_timer_ptr, 0, time_out);
      return;
}

void ssdbg2_stop_base_timer(void *base_timer_ptr)
 {
      stack_stop_timer((stack_timer_struct *)base_timer_ptr);
      return;
}

void ssdbg2_base_timer_timeout_hdler( void* msg_ptr )
{
      /* reset saved event id to avoid the potential bug to cancel released event */
      ssdbg2_context.ssdbg2_event_id = NULL;

      ssdbg2_context.ssdbg2_event_id = evshed_set_event(ssdbg2_context.ssdbg2_event_scheduler_ptr,
                                          (kal_timer_func_ptr)ssdbg2_base_timer_timeout_hdler, NULL, 200);
}
```

10 Memory Management

On MAUI, there is a variety of memory management schemes to meet the various requirements, they are known as control buffer, application dynamic memory (ADM), system specific memory and debug memory respectively. Among them, debug memory is needed if and only if DEBUG_KAL is defined, otherwise, it will be excluded if RELEASE_KAL is applied.

10.1 Descriptions

10.1.1 Control buffer

Control buffer is fixed size buffer, which offers not more than 2KB allocation, assertion failure “ctrl_buff_pool_info_g [pool_indx].pool_id” will be trapped in case of exceeding. It is internally inheritance from partitioned pool of Nucleus Plus, the predefined pool size are 8, 16, 32, 64, 128, 256, 512, 1024 and 2048Bytes. System always assign the smallest met control buffer to the user, for instance, if 50Bytes buffer is required, system will definitely allocate from 64B control buffer. Although space wasting, fragmentation issue is thoroughly eliminated.

In addition, system triggers fatal error “Buffer not available” (fatal error code 1 = 0x804, and error code 2 = size of control buffer), if the specific buffer size has been run out of free space, in other words, users of control buffer should be timing critical, resource sensitive, and may not have the capability of error handling.

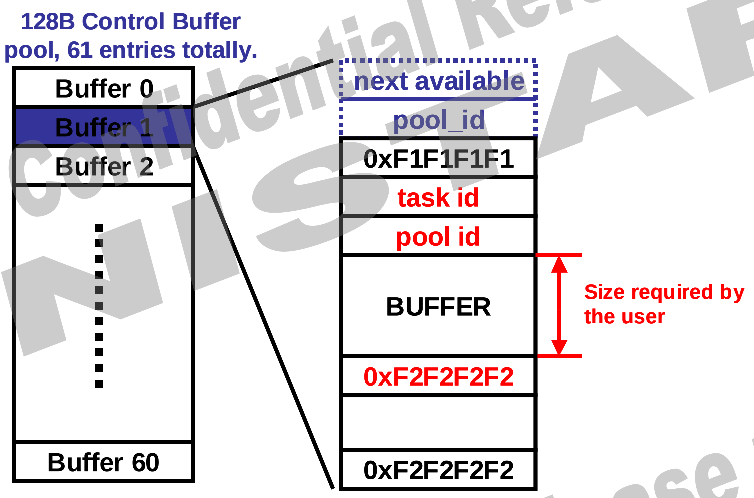
The control buffer size and number of each entry are defined in ***mcu\adaptation\include\ctrl_buff_pool.h***. However, customers could re-define the number of entries according to their requirement, except that, system error may occur if the original requirement is no longer satisfied (***mcu\custom\system\board version\custom_system.c***).

```
/* GPRS Class B Solution */
typedef enum {
  NUM_CTRL_BUFF_POOL_SIZE08   = 85,
  NUM_CTRL_BUFF_POOL_SIZE16   = 85,
  NUM_CTRL_BUFF_POOL_SIZE32   = 85,
  NUM_CTRL_BUFF_POOL_SIZE64   = 85,
  NUM_CTRL_BUFF_POOL_SIZE128  = 61,
  NUM_CTRL_BUFF_POOL_SIZE256  = 50,
  NUM_CTRL_BUFF_POOL_SIZE512  = 21,
  NUM_CTRL_BUFF_POOL_SIZE1024 = 17,
  NUM_CTRL_BUFF_POOL_SIZE2048 = 8,
  NUM_CTRL_BUFF_POOL_SIZE4096 = 0,
  NUM_CTRL_BUFF_POOL_SIZE8192 = 0,
  NUM_CTRL_BUFF_POOL_SIZE16384 =0,
  NUM_CTRL_BUFF_POOL_SIZE32768 =0,
  NUM_CTRL_BUFF_POOL_SIZE60000 =0
} ctrl_num_buff_pool_size;
```

```
/* GSM Only */
typedef enum {
  NUM_CTRL_BUFF_POOL_SIZE08   = 50,
  NUM_CTRL_BUFF_POOL_SIZE16   = 50,
  NUM_CTRL_BUFF_POOL_SIZE32   = 50,
  NUM_CTRL_BUFF_POOL_SIZE64   = 30,
  NUM_CTRL_BUFF_POOL_SIZE128  = 41,
  NUM_CTRL_BUFF_POOL_SIZE256  = 30,
  NUM_CTRL_BUFF_POOL_SIZE512  = 21,
  NUM_CTRL_BUFF_POOL_SIZE1024 = 17,
  NUM_CTRL_BUFF_POOL_SIZE2048 = 8,
  NUM_CTRL_BUFF_POOL_SIZE4096 = 0,
  NUM_CTRL_BUFF_POOL_SIZE8192 = 0,
  NUM_CTRL_BUFF_POOL_SIZE16384 =0,
  NUM_CTRL_BUFF_POOL_SIZE32768 =0,
  NUM_CTRL_BUFF_POOL_SIZE60000 =0
} ctrl_num_buff_pool_size;
```

```
void custom_config_ctrl_buff_info(void)
{
   /* These constants defined in adaptation\include\ctrl_buff_pool.h */
   ctrl_buff_pool_info_g[0].size        = CTRL_BUFF_POOL_SIZE08;
   ctrl_buff_pool_info_g[0].no_of_buff  = NUM_CTRL_BUFF_POOL_SIZE08;

   ctrl_buff_pool_info_g[1].size        = CTRL_BUFF_POOL_SIZE16;
   ctrl_buff_pool_info_g[1].no_of_buff  = NUM_CTRL_BUFF_POOL_SIZE16;

   ctrl_buff_pool_info_g[2].size        = CTRL_BUFF_POOL_SIZE32;
   ctrl_buff_pool_info_g[2].no_of_buff  = NUM_CTRL_BUFF_POOL_SIZE32;

   ctrl_buff_pool_info_g[3].size        = CTRL_BUFF_POOL_SIZE64;
   ctrl_buff_pool_info_g[3].no_of_buff  = NUM_CTRL_BUFF_POOL_SIZE64;

   ctrl_buff_pool_info_g[4].size        = CTRL_BUFF_POOL_SIZE128;
   ctrl_buff_pool_info_g[4].no_of_buff  = NUM_CTRL_BUFF_POOL_SIZE128;

   ctrl_buff_pool_info_g[5].size        = CTRL_BUFF_POOL_SIZE256;
   ctrl_buff_pool_info_g[5].no_of_buff  = NUM_CTRL_BUFF_POOL_SIZE256;

   ctrl_buff_pool_info_g[6].size        = CTRL_BUFF_POOL_SIZE512;
   ctrl_buff_pool_info_g[6].no_of_buff  = NUM_CTRL_BUFF_POOL_SIZE512;

   ctrl_buff_pool_info_g[7].size        = CTRL_BUFF_POOL_SIZE1024;
   ctrl_buff_pool_info_g[7].no_of_buff  = NUM_CTRL_BUFF_POOL_SIZE1024;

   ctrl_buff_pool_info_g[8].size        = CTRL_BUFF_POOL_SIZE2048;
   ctrl_buff_pool_info_g[8].no_of_buff  = NUM_CTRL_BUFF_POOL_SIZE2048;

   ctrl_buff_pool_info_g[9].size        = CTRL_BUFF_POOL_SIZE4096;
   ctrl_buff_pool_info_g[9].no_of_buff  = NUM_CTRL_BUFF_POOL_SIZE4096;

   ctrl_buff_pool_info_g[10].size       = CTRL_BUFF_POOL_SIZE8192;
   ctrl_buff_pool_info_g[10].no_of_buff = NUM_CTRL_BUFF_POOL_SIZE8192;

   ctrl_buff_pool_info_g[11].size       = CTRL_BUFF_POOL_SIZE16384;
   ctrl_buff_pool_info_g[11].no_of_buff = NUM_CTRL_BUFF_POOL_SIZE16384;

   ctrl_buff_pool_info_g[12].size       = CTRL_BUFF_POOL_SIZE32768;
   ctrl_buff_pool_info_g[12].no_of_buff = NUM_CTRL_BUFF_POOL_SIZE32768;

   ctrl_buff_pool_info_g[13].size       = CTRL_BUFF_POOL_SIZE65536;
   ctrl_buff_pool_info_g[13].no_of_buff = NUM_CTRL_BUFF_POOL_SIZE60000;
}
```

For easy debugging, some detection mechanisms are associated on each buffer entry, and they are isolated with compile options DEBUG_KAL and DEBUG_BUF. Figure below shows the internal structure of a buffer entry.



Internal structure of a buffer entry if DEBUG_KAL, DEBUG_BUF and DEBUG_BUF2 are defined

Corruption of either buffer header (0xF1F1F1F1) or footer (0xF2F2F2F2) will block the normal execution, and system error will be awakening instead. Detection of buffer corruption is done during buffer allocation and de-allocation.

Besides, MAUI provides an easy approach to diagnose memory leak issue. By monitoring operations on each buffer entry from specific control buffer size, user could easily determine whether the system is suffering from memory leak. Buffer monitoring would not work if either of the compile options DEBUG_KAL, DEBUG_BUF and DEBUG_BUF2 is excluded. Penalty of monitoring is more memory consumed and longer latency in allocating and releasing buffer, because task ID, file name, line number and buffer size will be recorded onto buffer history each time operation is done. Henceforth, buffer monitoring is officially turned-off.

To turn-on buffer monitoring at run-time, we need to know the ID of each control buffer size, they are denoted in Table 11-1.

Number	Size of control buffer	ID
0	8	1
1	16	2
2	32	4
3	64	8
4	128	16
5	256	32
6	512	64
7	1024	128
8	2048	256

Table 11-1. Identity number of control buffer

For example, to monitor 8 and 128Bytes control buffer, pressing the string in dialing screen, ***035670766*001*17#**. Be very careful that, system performance will definitely be scarified under the circumstance. Such enabling is not permanently available; it is reset in next time power-on

10.1.2 ADM

In addition to control buffer, upper layer applications may need more flexible dynamic memory mechanism, such as, allocating buffer larger than 2KB, going through its own error handling if allocation failure. MAUI provides an alternative dynamic memory management scheme, namely Application Dynamic Memory (ADM) to fulfill such requirement.

Users are responsible to provide a memory pool, and transfer the management to system; garbage collection is excluded from ADM. Specially note that, control block and others overhead for management will be created from the memory pool provided by users; therefore the memory pool size should take them into account. The minimal allocatable size is 8Bytes, and system guarantees the 4B alignment starting address for each allocation.

By using the sub-pool concept, system attempts to avoid fragmentation issue as much as possible. Under the architecture, all of the freed memory segments are categorized according to the pre-defined sub-pool size. It is user's responsibility to provide an appropriate sub-pool list. Internally, sub-pool is an array of type kal_unt32, and must be ended with 0xffffffff and 0x00. If it is set to NULL, default sub-pool setting will be applied.

```
kal_uint32 ADM_subpool_size[ ] = { 8, 16, 32, 64, 128, 0xffffffff, 0x00};
```

10.1.3 System memory and debug memory

Typically, system memory is a static memory pool declared at ***mcu\custom\system\board version\ custom_config.c***, it aims to provide the semi-static memory space for run-time usage. The term semi-static means that, the memory size is application dependent, but it would never be returned once allocated. For examples, control block of a task, task's stack, control block of a control buffer, buffer pool etc. Although system memory is restricted for system usage only, customers are responsible to provide its actual size since it is application dependent.

```
#define GLOBAL_MEM_SIZE              (300*1024)
#define GLOBAL_DEBUG_MEM_SIZE    (64*1024)

/* Use static array to check memory usage at compile time with corresponding scatter file. */
static kal_uint32 System_Mem_Pool[GLOBAL_MEM_SIZE/sizeof(kal_uint32)];
static kal_uint32 Debug_Mem_Pool[GLOBAL_DEBUG_MEM_SIZE/sizeof(kal_uint32)];
```

Tables below are components list of system and debug memory

Components	Size (In terms of Bytes)	
	RELEASE_KAL	DEBUG_KAL
kal_task_type	168	172
Task's stack	Task dependent, minimal requirement 240B.	Task dependent, minimal requirement 240B.
kal_hisr_type	88	92

HISR's stack	HISR dependent, minimal requirement 240B.	HISR dependent, minimal requirement 240B.
kal_queue_type	76	80
Queue pool	(Size of **ilm_struct**) x number of queue entries	(Size of **ilm_struct**) x number of queue entries
kal_mutex_type	40	48
kal_sem_type	48	52
kal_eventgrp_type	36	36
kal_timer_type	80	80
event_scheduler	68	68
kal_pool_type	60	72 + 52
Kal_internal_pool_statistics	0	16
Buffer pool	Total number of entries x [(4x2)+buffer size]	Total number of entries x [(4x6)+buffer size

Table 11-2. Occupiers of system memory

Components	Size (In terms of size)	Remark
kal_buff_stat_type	92	This is storage of buffer historical records, and **each buffer entry** occupies 92B.
kal_queue_stat_type	4	
kal_mutex_stat_type	8	
kal_timer_stat_type	12	

Table 11-3. Occupiers of debug memory

To meet the timing critical requirements, system memory is further divided into internal system memory and system memory; the former is linked at internal SRAM, whilst the later at external SRAM. Since the internal SRAM is too limited, the internal system memory is fixed for each base-band chip.

Before W04.41, system memory configuration follows convention illustrated in Table 11-4. Otherwise, Table 11-5 will be followed.

Chip / Protocol / Dynamic Memory	MT6205B	Others	
	GSM	ClassB	GSM
System Memory (External RAM)	300K	560K	430K
Internal System Memory (Internal RAM)	9.5K	16K	16K
Debug Memory	64K	128K	100K

Table 11-4. Internal system memory size configuration (before W04.41)

Dynamic Memory / Protocol / Chip	MT6205B	Others	
	GSM	ClassB	GSM
System Memory (External RAM)	150K	280K	256K
Internal System Memory (Internal RAM)	9.5K	16K	16K
Debug Memory	50K	100K	80K

Table 11-5. Internal system memory size configuration (W04.41 and later)

10.2 Data Structures and Data Types

10.2.1 Control buffer

Tables below illustrate the data structures and data types commonly used in managing control buffer.

kal_pool_statistics_struct, *kal_pool_statistics *(mcu\kal\include\kal_debug.h)*

Description:
Defines the statistical content of certain size of control buffer, the record is valid if and only if compile option DEBUG_BUF is defined.

Data Type	Element	Description
kal_uint16	num_buffs	Total number of the buffer entries, which is consistent with the configured number.
kal_uint16	buff_size	Buffer size in terms of Bytes, which is consistent with the configured number.
kal_uint16	current_allocation	Total number of the presently allocated entries.
kal_uint16	max_num_allocated	Tracking the maximal number of the allocated entries that system ever experienced.
kal_uint16	max_size_requested	Tracking the maximal allocated buffer size.

kal_history_node_t *(mcu\kal\common\include\kal_ debug common_defs. h)*

Description:
The physical data structure for buffer history, available if and only if compile option DEBUG_BUF is defined.

Data Type	Element	Description
kal_buff_state	buffer_state	Buffer status, either KAL_BUFF_ALLOCATED or KAL_BUFF_DEALLOCATED.
kal_internal_taskid	owner_task	Owner of the buffer, expressed with task identity.
kal_char*	file_name	File name.
kal_uint32	line	Line number.
kal_uint16	size	Size to be allocated.

kal_buff_stat_type, * kal_internal_buff_statistics *(mcu\kal\common\include\kal_ debug common_defs. h)*

Description:
Used for tracking the historical operations on a dedicated buffer entry. Again, it is valid if and only if compile option DEBUG_BUF is defined.

Data Type	Element	Description
kal_internal_taskid	owner_task	Owner of the buffer, expressed with task identity.
kal_uint8	buffer_state	Buffer status, either KAL_BUFF_ALLOCATED or KAL_BUFF_DEALLOCATED.
kal_history_node_t	buff_history [KAL_MAX_BUFF_HISTORY]	Array used for recording the operations ever processed and is associated on each buffer entry.

kal_pool_stat_type, *kal_internal_pool_statistics *(mcu\kal\include\kal_debugh)*

Description:

Defines statistical, as well as the historical data of a control buffer.

Data Type	Element	Description
kal_pool_statistics_struct	pool_info	Physical storage for buffer statistical record.
kal_internal_buff_statistics	buff_stat	Pointer to buffer historical logging.

kal_pool_type, *kal_internal_poolid *(mcu\kal\nucleus\include\kal_nucleus.h)*

Description:

Defines the control block of a control buffer, slightly different if DEBUG_KAL, DEBUG_BUF and DEBUG_BUF2 are defined (light yellow background color).

Data Type	Element	Description
kal_os_pool_type	pool_id	Pool identity inherits from partitioned pool of Nucleus Plus.
kal_bool	pool_debug_mask	KAL_TRUE if buffer monitoring is enabled, otherwise, disabled.
kal_uint16	buff_size;	Control buffer size in units of Bytes.
kal_internal_pool_statistics	pool_stat	Pointer to statistical data of the control buffer.
kal_mutexid	protecting_mutex	MUTEX used for protecting operations taken on dedicated control buffer.

buff_hdr_t *(mcu\kal\nucleus\include\kal_nucleus.h)*

Description:

Defines the structure of KAL buffer header, please refer to Figure 2 for the exact layout. It is valid if and only if DEBUG_KAL, DEBUG_BUF and DEBUG_BUF3 are defined.

Data Type	Element	Description
kal_uint32	hdr_stamp	KAL buffer header stamp, 0xF1F1F1F1.
kal_internal_taskid	task_id	Owner of the buffer entry, expressed as task identify.
kal_internal_poolid	pool_id	Control block to which the buffer belongs.
kal_uint8	usr_buff[1]	Starting address of the buffer pool.

buff_pool_info_struct *(mcu\adaptation\include\stack_buff_pool.h)*

Description:

Defines the control block of message queue, slightly different if DEBUG_KAL and DEBUG_ITC are defined (light yellow background color).

Data Type	Element	Description
kal_poolid	pool_id	Pool identify assigned by the system during creation.
kal_uint32	size	Control buffer size.
kal_uint16	no_of_buff	Total number of buffer entries.
kal_uint32	num_of_misses	Total number of misses; for control buffer, it is always zero, because system would never return NULL pointer on MAUI.

Data Types	Description
kal_poolid	Identity of a control buffer, which is internally equivalent to kal_internal_poolid.

kal_buff_state	Enumeration type of buffer status.
	typedef enum {
	  KAL_BUFF_DEALLOCATED,
	  KAL_BUFF_ALLOCATED,
	  KAL_BUFF_CORRUPTED
	} kal_buff_state;

Global variable	Description
ctrl_buff_pool_info_g	Array of **buff_pool_info_struct,** number of the entries is determined by RPS_CREATED_CTRL_BUFF_POOLS defined in ***mcu\adaptation\include\ctrl_buff_pool.h***. It is an unified entrance of control buffer.

10.2.2 ADM

struct ADM_MB_HEAD_STRUCT, ADM_MB_HEAD *(mcu\kal\common\include\kal_adm.h)*

Description:
Defines the header structure of an ADM.

Data Type	Element	Description
struct ADM_MB_HEAD_STRUCT *	prev	Pointer to the previous memory block (MB).
struct ADM_MB_HEAD_STRUCT *	next	Pointer to the next memory block (MB).
struct ADM_MB_HEAD_STRUCT *	bl_prev	Pointer to the previous block list (BL).
struct ADM_MB_HEAD_STRUCT *	bl_next	Pointer to the next block list (BL).

ADM_MB_FOOT *(mcu\kal\common\include\kal_adm.h)*

Description:
Defines the footer of a memory piece allocated from some ADM.

Data Type	Element	Description
kal_uint32	stamp	Fixed pattern 0x04F4F4F4 for the detection of memory corruption.

ADM_MB_LOG *(mcu\kal\common\include\kal_adm.h)*

Description:
Defines the data structure of ADM memory logging.

Data Type	Element	Description
kal_uint32	stamp	Fixed pattern 0x03F3F3F3 for the detection of memory corruption.
kal_char *	filename	File name.
kal_uint32	line	Line number.

ADM_CB *(mcu\kal\common\include\kal_adm.h)*

Description:
Defines the control block of an ADM.

Data Type	Element	Description

kal_uint32	adm_id	Identity of ADM.
kal_uint16	bl_num	Number of block lists.
kal_uint16	owner	Owner of the ADM, existed if DEBUG_ADM is defined, otherwise, it is a reserved field.
kal_uint8	islogging	KAL_TRUE if buffer logging is enabled.
kal_uint8	reserved2[3]	Reserved field for alignment.

Data Types	Description
KAL_ADM_ID	Identity of an ADM, it is type of void *.

10.3 Memory management APIs

10.3.1 Control buffer

get_ctrl_buffer

Prototype:	void * get_ctrl_buffer (size)
Header file:	app_alloc_buff.h
Input:	*size* is size of buffer to be allocated.
Output:	Buffer pointer is returned in the manner of void *, any unsuccessful operation will be re-directed to system error.
Description:	This is common interface for allocating a buffer from the smallest fit control buffer.

free_ctrl_buffer

Prototype:	void free_ctrl_buffer (void * ptr)
Header file:	app_alloc_buff.h
Input:	*ptr* is buffer pointer to be released.
Description:	This is common interface for returning a buffer pointer to the system.

kal_query_ctrlbuf_max_consumption

Prototype:	kal_bool kal_query_ctrlbuf_max_consumption (kal_uint32 *ptr)
Header file:	kal_release.h
Input:	*ptr* is an array, which is used to return values.
Output:	KAL_FALSE if DEBUG_KAL is not defined, otherwise KAL_TRUE.
Description:	This service is provided for querying currently maximum consumption on each control buffer, which is totally RPS_CREATED_CTRL_BUFF_POOLS.

10.3.2 ADM

kal_adm_create

Prototype:	KAL_ADM_ID kal_adm_create (void *mem_addr, kal_uint32 size, kal_uint32 *subpool_size, \ kal_bool islogging)
Header file:	kal_release.h
Input:	*mem_addr* is starting address of the memory pool, *size* is total size of the memory pool, *subpool_size* is an array, which describe the total number of sub-pools and its size, *islogging* is logging flag, guard pattern checking and operation logging will be enabled if it is KAL_TRUE.
Output:	ID the created ADM.
Description:	This service aims for creating an ADM from the given memory pool, ADM control block, header and footer footprints must be taken into account while calculating the pool size. Total overhead is the summation of the following, Fixed size overhead, 12 + 16 + 16 x (number of sub-pool entries). Floating overhead, 8B header for each allocation, extra 16B overhead for footer if logging is enabled. Default value will be adopted if *subpool_size* is NULL.

kal_adm_delete

Prototype:	kal_status kal_adm_delete (KAL_ADM_ID adm_id)
Header file:	kal_release.h

Input:	*adm_id* is identity of an ADM to be deleted.
Output:	KAL_ADM_SUCCEED if operation successfully done, KAL_MEMORY_NOT_RELEASE if there are some memory block not yet returned.
Description:	This function call is specific for deleting an ADM.

kal_adm_alloc

Prototype:	void * kal_adm_alloc (adm_id, size)
Header file:	kal_release.h
Input:	*adm_id* is destination ADM, from where a memory piece will be allocated, and *size* is the required memory size.
Output:	Pointer of the allocated memory piece if operation successfully done, otherwise, NULL.
Description:	This service aims at allocating a memory piece from the dedicated ADM. Be very careful that, if ADM logging is enabled, extra memory space will be consumed. It is excluded from the *size* specified in input parameter.

kal_adm_free

Prototype:	void kal_adm_free (KAL_ADM_ID adm_id, void *mem_addr))
Header file:	kal_release.h
Input:	*adm_id* is destination ADM, to which the memory pointer *mem_addr* will be returned.
Description:	This function call aims at releasing a memory pointer to the dedicated ADM; if ADM logging is turned on, memory header and footer will be checked during pointer releasing.

kal_adm_get_max_alloc_size

Prototype:	kal_uint32 kal_adm_get_max_alloc_size (KAL_ADM_ID adm_id)
Header file:	kal_release.h
Input:	*adm_id* is destination ADM to be queried.
Output:	Maximal available memory size to be allocated.
Description:	This function is designed for querying the maximal available memory size that ADM is affordable. If ADM logging is turned-on, the actual exercisable size should subtract the logging (size of (ADM_MB_LOG)), header (size of ()) and footer size (size of (ADM_MB_FOOT)).

kal_adm_get_total_left_size

Prototype:	kal_uint32 kal_adm_get_total_left_size (KAL_ADM_ID adm_id)
Header file:	kal_release.h
Input:	*adm_id* is destination ADM to be queried.
Output:	Total free space of the dedicated ADM.
Description:	Users could rely on this service for retrieving total free space of dedicated ADM; it is very useful in identifying fragmentation.

kal_adm_check_integrity

Prototype:	void *kal_adm_check_integrity (KAL_ADM_ID adm_id)
Header file:	kal_release.h
Input:	*adm_id* is destination ADM to be queried.
Output:	Address of corrupted memory block. If there is no memory corruption, NULL is returned.
Description:	This function is only available from 05C W06.17. This function ensures the integrity of all allocated memory blocks. If the header or footer of any allocated memory block is corrupted, it returns the address of the corrupted memory block. This is useful for debugging memory corruption.

10.3.3 System and debug memory

kal_sys_mem_query_freesize

Prototype:	kal_uint32 kal_sys_mem_query_freesize (void)
Header file:	kal_release.h
Output:	An integer value of available memory space.
Description:	This function call is designed for querying the system memory free space.

kal_debug_mem_query_freesize

Prototype:	kal_uint32 kal_debug_mem_query_freesize (void)
Header file:	kal_release.h
Output:	An integer value of available memory space.
Description:	This function call is designed for querying the debug memory free space; note that, it always returns zero if DEBUG_KAL is not defined.

10.4 Examples

Below are two examples of ADM.

(A) Without using sub-pool

```
  #include “kal_release.h”

  static kal_uint8  my_heap[3 * 1024];
  KAL_ADM_ID   my_dm_id;
  kal_uint32 *ptr;

/* create ADM */
my_dm_id = kal_adm_create(my_heap, 3 * 1024, NULL, KAL_FALSE);
if (my_dm_id == NULL)
  my_error_handler();   /* fail to create a dm pool */


/* allocate memory */
ptr = (kal_uint32 *)kal_adm_alloc(my_dm_id, 1024);
if (ptr == NULL) {
  if (kal_adm_get_total_left_size(my_dm_id) > 1024)
    my_error_handler();   /* fragmentation */
  else
    my_error_handler();   /* out of memory */
}

/* free memory */
kal_adm_free(my_dm_id, ptr);

/* delete ADM */
if (kal_adm_delete(my_dm_id) != KAL_ADM_SUCCEED)
  my_error_handler();
```

(A) With sub-pool

```
#include “kal_release.h”

static kal_uint32 my_subpool_size[] = { 8, 16, 24, 32, 40, 48, 56, 64, 72, 80, 88,
                                        96, 104, 112, 120, 128, 0xffffffff, 0 };
static kal_uint8 my_heap[3 * 1024];
KAL_ADM_ID   my_dm_id;
kal_uint32 *ptr;

/* create ADM */
my_dm_id = kal_adm_create(my_heap, 3 * 1024, my_subpool_size, KAL_FALSE);
if (my_dm_id == NULL)
  my_error_handler();   /* fail to create a dm pool */
```

11 Utility APIs

This chapter collects all the utility APIs, which may be an inline function, macro or the re-written APIs to avoid re-entrance problem.

11.1 Memory operation

kal_mem_cmp

Prototype:	kal_int32 kal_mem_cmp (void* src1, void* src2, kal_uint32 size)
Header file:	kal_release.h
Description:	It is an inline function of mem_cmp; it compares the first *size* Bytes of the arrays pointed to by *src1* and *src2*.

kal_mem_cpy

Prototype:	void * kal_mem_cpy (void* dest, const void* src, kal_uint32 size)
Header file:	kal_release.h
Description:	It is an inline function of mem_cpy; it copies *size* Bytes from the array pointed to by *src* into the array pointed to by *dest*. If the arrays overlap, the behavior is undefined.

kal_mem_set

Prototype:	void * kal_mem_set (void* dest, kal_uint8 value, kal_uint32 size)
Header file:	kal_release.h
Description:	It is an inline function of mem_set; it copies *value* into the first *size* Bytes of the array pointed to by *dest*, it returns *dest*.

11.2 Boot mode querying

stack_query_boot_mode

Prototype:	boot_mode_type stack_query_boot_mode (void)
Header file:	kal_release.h
Output:	FACTORY_BOOT if they system is in META or factory mode, USBMS_BOOT if the system is in USB boot mode, NORMAL_BOOT if the system is in normal power-on mode, UNKNOWN_BOOT_MODE if the system not yet known its boot-mode.
Description:	This service aims at providing boot mode information to caller, it is available for LISR/HISR/task level query, in system boot-up stage, UNKNOWN_BOOT_MODE will probably get.

11.3 Multi-bytes string processing

kal_dchar_strlen

Prototype:	int kal_dchar_strlen (const char *wstr)
Header file:	kal_release.h
Input:	*wstr* is pointer of string.
Output:	Length of a double character string, in terms of Bytes.

Description: This function returns the length of the null-terminated double character string pointed to by *wstr*. For double character string, null-terminator is 0x00 0x00, and are not counted.

kal_dchar_strcpy

Prototype:	char *kal_dchar_strcpy (char *to, const char *from)
Header file:	kal_release.h
Input:	*to* and *from* are destination and source string respectively.
Output:	The double characters string being copied.
Description:	This function call aims to copy the content of double characters string *from* into double characters string *to*.

kal_dchar_strncpy

Prototype:	char *kal_dchar_strncpy (char *to, const char *from, int n)
Header file:	kal_release.h
Input:	*to* and *from* are destination and source double characters string respectively, *n* is total number of wide characters to be copied.
Output:	The double characters string being copied.
Description:	This function copies up to *n* wide characters from the double characters string pointed to by *from* into the array pointed to by *to*.

kal_dchar_strcmp

Prototype:	int kal_dchar_strcmp (const char *s1, const char *s2)
Header file:	kal_release.h
Input:	*s1* and *s2* are two double characters strings to be compared.
Output:	0 if the two double characters strings are exactly identical, <0 if s1 is less than s2, >0 if s1 is greater than s2.
Description:	This function lexicographically compares two double characters strings.

kal_dchar_strncmp

Prototype:	int kal_dchar_strncmp (const char *s1, const char *s2, int n)
Header file:	kal_release.h
Input:	*s1* and *s2* are two double characters strings to be compared, *n* is total number of wide characters to be compared.
Output:	0 if the two double characters strings are exactly identical, <0 if s1 is less than s2, >0 if s1 is greater than s2.
Description:	This function lexicographically compares two double characters strings not more than *n* wide characters.

kal_dchar_strcat

Prototype:	char *kal_dchar_strcat(char *s1, const char *s2)
Header file:	kal_release.h
Input:	*s1* and *s2* are two double characters strings to be concatenated.
Output:	Double characters string being concatenated.
Description:	This function concatenates a copy of *s2* to *s1* and terminates *s1* with a 0x00 0x00.The null terminator 0x0000 originally ending *s1* is overwritten by the first character of *s2*.

kal_dchar_strncat

Prototype:	char *kal_dchar_strncat (char *s1, const char *s2, int n)
Header file:	kal_release.h
Input:	*s1* and *s2* are two double characters strings to be concatenated, this action will be taken on *n* wide characters.
Output:	Double characters string being concatenated.
Description:	This function concatenates not more than *n* wide characters of the double characters string pointed to by *s2* to the double characters string pointed to by *s1* and terminates *s1* with 0x0000. The null terminator 0x00 0x00 originally ending *s1* is overwritten by the first character of *s2*.

kal_dchar_strchr

Prototype:	char *kal_dchar_strchr (const char *s, int c)
Header file:	kal_release.h
Input:	*s* is pointer of double characters string, *c* is wide character to be searched.
Output:	Double characters string.
Description:	This function returns a pointer to the first occurrence of the *c* in the double characters string pointed to by *s*.

kal_dchar_strrchr

Prototype:	char *kal_dchar_strrchr (const char *str, int ch)
Header file:	kal_release.h
Input:	*str* is pointer of double characters string, *ch* is wide character to be searched.
Output:	Double characters string.
Description:	This function returns a pointer to the last occurrence of the *ch* in the double characters string pointed to by *str*.

kal_dchar2char

Prototype:	void kal_dchar2char (WCHAR *outstr, char* tostr)
Header file:	kal_release.h
Input:	*outstr* is pointer of wide character string, and *tostr* is double character string.
Description:	This function is used to convert a wide character string to a single character string.

kal_wsprintf

Prototype:	void kal_wsprintf (WCHAR *outstr, char *fmt,...)
Header file:	kal_release.h
Description:	Identical to sprintf, except that input string is a wide character string.

kal_wstrlen

Prototype:	int kal_wstrlen (const WCHAR *wstr)
Header file:	kal_release.h
Input:	*wstr* is pointer of wide character string.
Output:	Length of a wide character string, in terms of WCHAR..
Description:	This function returns the length of the null-terminated string pointed to by *wstr*. For wide character string, null-terminator is 0x0000, and is not counted.

kal_wstrcpy

Prototype:	WCHAR *kal_wstrcpy (WCHAR *to, const WCHAR *from)

Header file:	kal_release.h
Input:	*to* and *from* are destination and source wide character string respectively.
Output:	The wide character string being copied.
Description:	This function call aims to copy the content of wide character string *from* into wide character string *to*.

kal_wstrncpy

Prototype:	WCHAR *kal_wstrncpy (WCHAR *to, const WCHAR *from, int n)
Header file:	kal_release.h
Input:	*to* and *from* are destination and source wide character string respectively, *n* is total number of wide characters to be copied.
Output:	The wide character string being copied.
Description:	This function copies up to *n* wide characters from the wide character string pointed to by *from* into the array pointed to by *to*.

kal_wstrcmp

Prototype:	int kal_wstrcmp (const WCHAR *s1, const WCHAR *s2)
Header file:	kal_release.h
Input:	*s1* and *s2* are two wide character strings to be compared.
Output:	0 if the two wide character strings are exactly identical, <0 if s1 is less than s2, >0 if s1 is greater than s2.
Description:	This function lexicographically compares two wide character strings.

kal_wstrncmp

Prototype:	int kal_wstrncmp (const WCHAR *s1, const WCHAR *s2, int n)
Header file:	kal_release.h
Input:	*s1* and *s2* are two wide character strings to be compared, *n* is total number of wide characters to be compared.
Output:	0 if the two wide character strings are exactly identical, <0 if s1 is less than s2, >0 if s1 is greater than s2.
Description:	This function lexicographically compares two wide character strings not more than *n* wide characters.

kal_wstrcat

Prototype:	WCHAR *kal_wstrcat(WCHAR *s1, const WCHAR *s2)
Header file:	kal_release.h
Input:	*s1* and *s2* are two wide character strings to be concatenated.
Output:	Wide character string being concatenated.
Description:	This function concatenates a copy of *s2* to *s1* and terminates *s1* with a 0x0000.The null terminator 0x0000 originally ending *s1* is overwritten by the first character of *s2*.

kal_wstrncat

Prototype:	WCHAR *kal_wstrncat (WCHAR *s1, const WCHAR *s2, int n)
Header file:	kal_release.h
Input:	*s1* and *s2* are two wide character strings to be concatenated, this action will be taken on *n* wide characters.
Output:	Wide character string being concatenated.
Description:	This function concatenates not more than *n* wide characters of the wide character string pointed to by *s2* to the wide character string pointed to by *s1* and terminates *s1* with 0x0000. The null terminator 0x0000 originally ending *s1* is overwritten by the first character of *s2*.

kal_wstrchr

Prototype:	WCHAR *kal_wstrchr (const WCHAR *s, int c)
Header file:	kal_release.h
Input:	*s* is pointer of wide character string, *c* is wide character to be searched.
Output:	Wide character string.
Description:	This function returns a pointer to the first occurrence of the *c* in the wide character string pointed to by *s*.

kal_wstrrchr

Prototype:	WCHAR *kal_wstrrchr (const WCHAR *str, int ch)
Header file:	kal_release.h
Input:	*str* is pointer of wide character string, *ch* is wide character to be searched.
Output:	Wide character string.
Description:	This function returns a pointer to the last occurrence of the *ch* in the wide character string pointed to by *str*.

11.4 Reentrance functions

kal_strtok_r

Prototype:	kal_char * kal_strtok_r (kal_char *string, const kal_char *seperators, kal_char **ppLast)
Header file:	kal_release.h
Input:	*string* is string to be tokenized, *separators* tells the separator, while *ppLast* serves as string index.
Output:	Pointer to the first character of a token
Description:	This is a re-entrance function, where more than one caller could tokenize the same source string with individual string index.

kal_gmtime_r

Prototype:	struct tm *kal_gmtime_r (const time_t *timer, struct tm *t)
Header file:	kal_release.h
Input:	*timer* is time to be converted, *t* is pointer for returning the conversion result.
Output:	The broken-down form of timer, it is same with *t*.
Description:	This is a re-entrance function used for returning the broken-down form of *timer* in the form of a *tm* structure.

11.5 Exception handling

ASSERT

Prototype:	ASSERT(expression)
Header file:	kal_release.h
Input:	*expression* is an expression.
Description:	If DEBUG_KAL is defined, ASSERT takes action if the *expression* evaluates to FALSE, the embedded exception handler will then be invoked (please refer to ExceptionHandling.pdf for more detailed). Otherwise, in RELEASE_KAL, ASSERT does nothing.

EXT_ASSERT

Prototype:	EXT_ASSERT(expr, e1, e2, e3)
Header file:	kal_release.h

Input: *expression* is an expression, *e1*, *e2* and *e3* are three extended parameters with type of kal_uint32.
Description: Unlike ASSERT, EXT_ASSERT always take action no matter DEBUG_KAL or RELEASE_KAL; in addition to expression, users are given three extended parameters for tracking the violations.

EXT_ASSERT_DUMP

Prototype: EXT_ASSERT_DUMP(expr, e1, e2, e3, dump_param)
Header file: kal_release.h
Input: *expression* is an expression, *e1*, *e2* and *e3* are three extended parameters with type of kal_uint32.
dump_param is a pointer to a ASSERT_DUMP_PARAM_T structure.
Description: EXT_ASSERT_DUMP is just like EXT_ASSERT except that it can dump at most 10 memory fragments in the exception log. (224 bytes is reserved in the exception log.) Below is the ASSERT_DUMP_PARAM_T structure:

```
typedef struct ASSERT_DUMP_PARAM
{
   kal_uint32 addr[ASSERT_DUMP_PARAM_MAX];
   kal_uint32 len[ASSERT_DUMP_PARAM_MAX];  /* in bytes */
} ASSERT_DUMP_PARAM_T;
```

Specify 10 memory fragments in the structure as below:

```
ASSERT_DUMP_PARAM_T dump_param;

dump_param.addr[0] = 0x08000000;
dump_param.len[0] = 8;
dump_param.addr[1] = 0x08000000 + 100;
dump_param.len[1] = 8;
dump_param.addr[2] = 0x08000000 + 200;
dump_param.len[2] = 8;
dump_param.addr[3] = 0x08000000 + 300;
dump_param.len[3] = 8;
dump_param.addr[4] = 0x08000000 + 400;
dump_param.len[4] = 8;
dump_param.addr[5] = 0x08000000 + 500;
dump_param.len[5] = 8;
dump_param.addr[6] = 0x08000000 + 600;
dump_param.len[6] = 8;
dump_param.addr[7] = 0x08000000 + 700;
dump_param.len[7] = 8;
dump_param.addr[8] = 0x08000000 + 800;
dump_param.len[8] = 8;
dump_param.addr[9] = 0x08000000 + 900;
dump_param.len[9] = 0x100000;

EXT_ASSERT_DUMP(0, 1, 2, 3, &dump_param);
```

When the exception handler processes the exception later, your data will be stored in the exception log. This may be useful for developers to debug. Please note that a NULL address must follow your last specified address. For example, if you only want to dump 5 memory fragments, the 6-th entry of the „addr" field must be NULL (dump_param.addr[5] = NULL).