Freescale Semiconductor, Inc.
Reference Manual

Document Number: CONNFWKRM
Rev .4,
03/2015

Connectivity Framework

Reference Manual

1 Introduction

The scope of this document is the Connectivity Framework software used to ensure portability across the ARM-based microcontrollers portfolio of the Freescale connectivity stacks.

1.1 Audience

This document is primarily intended for internal software development teams, but its contents can be shared with customers or partners under the same licensing agreement as the framework software itself.

Contents

1.2 References

http://www.freescale.com/webapp/sps/site/homepage.jsp?code=KINETIS&tid=vanKINETIS

http://www.arm.com/products/processors/cortex-m/cortex-microcontroller-software-interface-standard.php

http://www.freescale.com/webapp/sps/site/homepage.jsp?code=MQX_HOME&tid=vanMQX

http://www.freertos.org/

1.3 Acronyms and Abbreviations

Table 1 Acronyms and Abbreviations

Acronym / Term	Description
FSCI	Freescale Serial Communication Interface
NV Storage	Non-Volatile Storage Subsystem
PHY	Physical Layer
MAC	Medium Access Control Layer
NWK	Network
API	Application Programming Interface
OS	Operating System
TMR	Timer
RNG	Random Number Generator
HAL	Hardware Abstraction Layer
USB	Universal Serial Bus
NVIC	Nester Vector Interrupt Controller
PWR	Power
RST	Reset
UART	Universal Asynchronous Receiver-Transmitter
SPI	Serial Peripheral Interface
I2C	Inter-Integrated Circuit
GPIO	General Purpose Input/Output

2 Overview

System architecture decomposition is depicted in the figure below. As can be observed, the framework, FSCI (Test Client) and the components are at same level, offering its services to the upper layers.

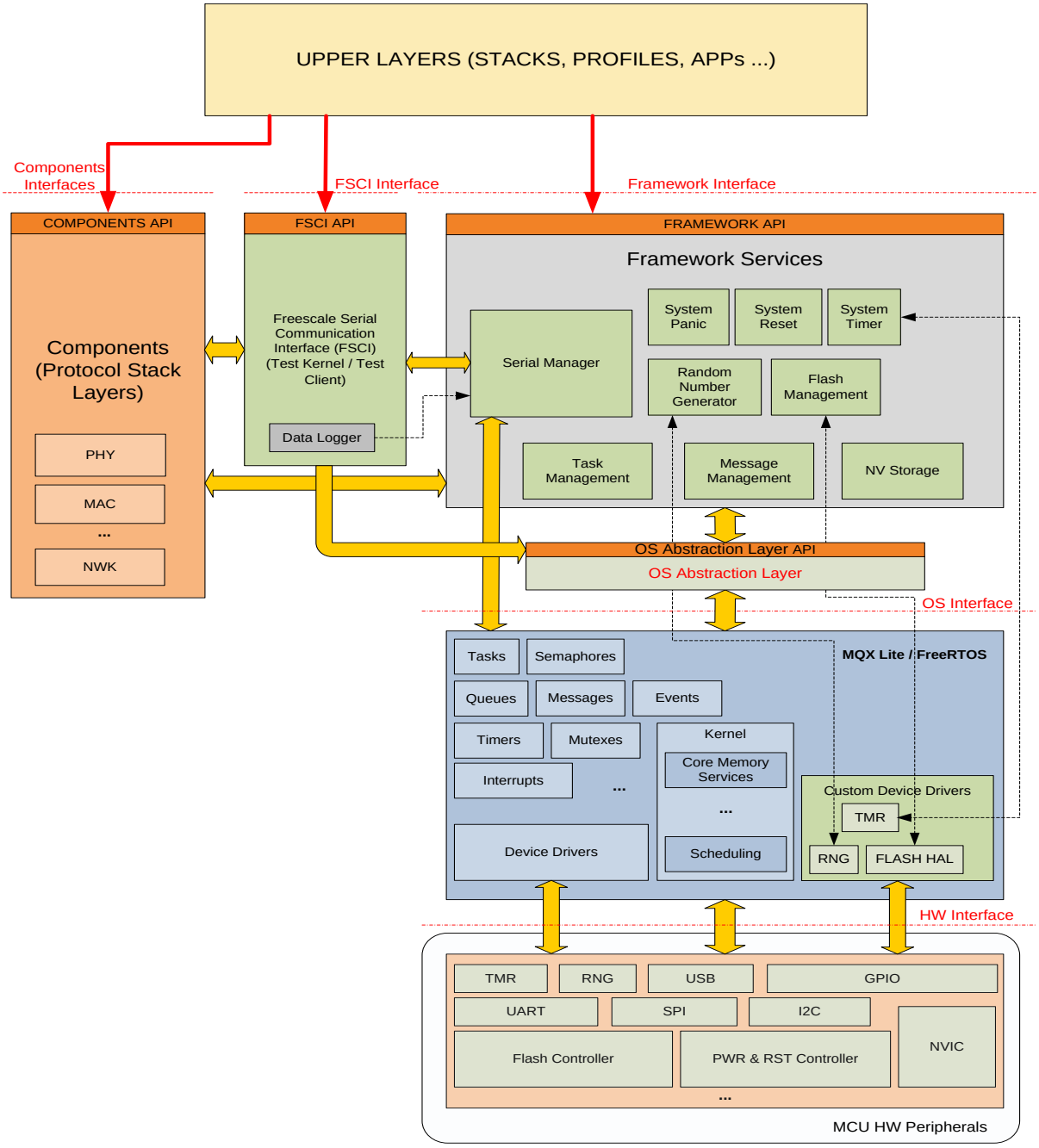

Figure 1. System Decomposition

All the framework services interact with the operating system through a so called OS Abstraction Layer. The role of this layer is to offer an “OS agnostic” separation between the operating system and all the upper layers. Detailed information about the framework services is presented in the following chapters.

3 Framework Services

3.1 OS Abstraction

3.1.1 Overview

The framework and other connectivity software modules that use RTOS services never use the RTOS API directly, instead they use the API exposed by the OS Abstraction. This ensures portability across multiple operating systems. If the use of an operating system which is not currently supported by the OS Abstraction layer is desired, an OS adapter must be implemented.

The OS adapter is a source file which contains wrapper functions over the RTOS API. This usually involves tweaking parameters and gathering some additional information not provided in the parameters. Sometimes more complex tasks must be performed if the functionality provided by the operating system is too different, for example the implementation of signals in FreeRTOS and timers in MQX.

To add support for another operating system all functions and macros detailed in this chapter must be implemented for each RTOS. In addition, all typedefs should be analyzed and modified if the need arises.

The purpose of the OS Abstraction layer is to remove dependencies of the stack and user code on a specific operating system. Because operating systems differ in services, APIs, data types etc. some restrictions and enhancements are needed within the OS Abstraction layer that reflects throughout the code.

This version of the OS Abstraction layer is implemented for MQX and FreeRTOS. The API provided in the connectivity framework is an extension of the OSA provided in KSDK.

NOTE

The OSA_EXT module was created to be used by the Connectivity software libraries. The use of the OSA_EXT API it is not recommended for applications developing.

3.1.2 Task Creation and Control

3.1.2.1 Overview

The OS Abstraction layer offers common task creation and control services for MQX, FreeRTOS, uCosII, uCosIII and Bare Metal. The OS Abstraction provides the following services for task creation and control:

- Create
- Terminate
- Wait
- Get ID
- Yield
- Set priority
- Get priority

In the OS Abstraction layer a task named main_task() is used as the starting point. The user must implement a function with the prototype *extern void main_task(uint32_t)* and treat it like a task. The OS Abstraction implementation declares this function as external.

From this task the user can create other tasks previously defined with *osThreadDef(name, priority, instances, stackSz)*. After system initialization, the *main_task* can either be terminated or reused. Please note that terminating main_task does not free the used memory since the task stack is a global array.

The *main_task* task initially has the lowest priority. If necessary, the priority can be modified at runtime using the *OSA_EXT_TaskSetPriority* API.

Some framework components require a task to be defined and created. The task is defined in the source files of the module and task creation is done in the initialization function. This approach makes the integration process easier, without adding extra steps to the initialization process.

Tasks can be defined using the *OSA_EXT_TASK_DEFINE* macro at compile time, and are not automatically started. After that, tasks can be created anytime the user wants to, using the *OSA_EXT_TaskCreate* API.

For MQX, task stacks are arrays defined by the *_EXT_TASK_DEFINE* macro and for FreeRTOS stacks are allocated internally.

Tasks may also have multiple instances. The code to be executed is the same for all instances but each instance has its own stack. When using multiple instances, the stack array is multiplied by the maximum number of instances. Tasks can also be terminated but please note that for MQX the task stack cannot be freed since it is a static array.

3.1.2.2 Constant Macro Definitions

Name:

```
#define OSA_PRIORITY_IDLE          (6)
#define OSA_PRIORITY_LOW           (5)
#define OSA_PRIORITY_BELOW_NORMAL  (4)
#define OSA_PRIORITY_NORMAL        (3)
#define OSA_PRIORITY_ABOVE_NORMAL  (2)
#define OSA_PRIORITY_HIGH          (1)
#define OSA_PRIORITY_REAL_TIME     (0)
#define OSA_TASK_PRIORITY_MAX      (0)
#define OSA_TASK_PRIORITY_MIN     (15)
```

Description:

Defines the priority levels used by the OSA_EXT

Name:

```
#define OSA_EXT_TASK_DEFINE (name, priority, instances, stackSz, useFloat)
```

Description:

It defines a task using the name as an identifier.

- priority – the task priority
- instances – the maximum number of instances the task may have
- stackSz – the task stack size in bytes
- useFloat – specifies if the task uses float operations or not!

Name:

```
#define OSA_EXT_TASK (name)
```

Description:

It is used to reference a thread definition by name.

Name:

```
#define osaWaitForever_c   ((uint32_t)(-1)) ///< wait forever timeout value
```

Description:

It is used to indicate an infinite wait period.

3.1.2.3 User defined data type definitions

Name:

```
typedef enum osaTimerDef_tag{
    osaStatus_Success = 0U,
    osaStatus_Error   = 1U,
    osaStatus_Timeout = 2U,
    osaStatus_Idle    = 3U
```

```
};
```

Description:

OSA EXT error codes.

Name:

```
typedef void (*osaTaskPtr_t) (osaTaskParam_t argument);
```

Description:

The data type definition for the task function pointer.

Name:

```
typedef void *osaTaskId_t;
```

Description:

The data type definition for the task ID. The value stored is different for each OS.

3.1.2.4 API Primitives

main_task ()

Prototype:

```
extern void main_task (uint32_t param);
```

Description:
Prototype of the user implemented *main_task.*

Parameters:

Name	Type	Direction	Description
param	Uint32_t	[IN]	Parameter passed to the task upon creation.

Returns:

None.

OSA_EXT_TaskCreate ()

Prototype:

```
osaTaskId_t OSA_EXT_TaskCreate(osaThreadDef_t *thread_def, osaTaskParam_t task_param);
```

Description:
Creates a thread and adds it to active threads and sets it to state READY.

Parameters:

Name	Type	Direction	Description
thred_def	osaThreadDef_t	[IN]	Definition of the task.
argument	osaTaskParam_t	[IN]	Parameter to pass to the newly created task.

Returns:

Thread ID for reference by other functions or NULL in case of error.

OSA_EXT_TaskGetId ()

Prototype:

```
osaTaskId_t OSA_EXT_TaskGetId(void);
```

Description:
Returns the thread ID of the calling thread.

Parameters:

None.

Returns:

ID of the calling thread.

OSA_EXT_TaskDestroy ()

Prototype:

```
osaStatus_t OSA_EXT_TaskDestroy(osaTaskId_t taskId);
```

Description:
Terminates the execution of a thread.

Parameters:

Name	Type	Direction	Description
taskId	osaTaskId_t	[IN]	ID of the task.

Returns:

- osaStatus_Success The task was successfully destroyed.
- osaStatus_Error Task destruction failed or invalid parameter.

OSA_EXT_TaskYield ()

Prototype:

```
osaStatus_t OSA_EXT_TaskYield(void);
```

Description:
Passes control to next thread that is in state READY.

Parameters:

None.

Returns:

- osaStatus_Success The function is called successfully.
- osaStatus_Error Error occurs with this function.

OSA_EXT_TaskSetPriority ()

Prototype:

```
osaStatus_t OSA_EXT_TaskSetPriority(osaTaskId_t taskId, osaTaskPriority_t taskPriority);
```

Description:
Changes the priority of the task represented by the given task ID.

Parameters:

Name	Type	Direction	Description
taskId	osaTaskId_t	[IN]	ID of the task.
taskPriority	osaTaskPriority_t	[IN]	The new priority of the task.

Returns:

- osaStatus_Success Task's priority is set successfully.
- osaStatus_Error Task's priority can not be set.

OSA_EXT_TaskGetPriority ()

Prototype:

```
osaTaskPriority_t OSA_EXT_TaskGetPriority(osaTaskId_t taskId);
```

Description:
Gets the priority of the task represented by the given task ID.

Parameters:

Name	Type	Direction	Description
taskId	osaTaskId_t	[IN]	ID of the task.

Returns:

Current priority value of the thread.

OSA_EXT_TimeDelay ()

Prototype:

```
void OSA_EXT_TimeDelay (uint32_t millisec);
```

Description:

Suspends the calling task for a given amount of milliseconds.

Parameters:

Name	Type	Direction	Description
millisec	uint32_t	[IN]	Amount of time to suspend the task.

Returns:

None.

Task Creation Example

```
OSA_EXT_TASK_DEFINE ( Job1, OSA_PRIORITY_HIGH,  1,   800, 0);
OSA_EXT_TASK_DEFINE ( Job2, OSA_PRIORITY_ABOVE_NORMAL, 2,   500, 0);

void main_task(void const *param)
{
  OSA_EXT_TaskCreate (OSA_EXT_TASK (Job1), (osaTaskParam_t)NULL);
  OSA_EXT_TaskCreate (OSA_EXT_TASK (Job2), (osaTaskParam_t)1);
  OSA_EXT_TaskCreate (OSA_EXT_TASK (Job2), (osaTaskParam_t)2);

  OSA_EXT_TaskDestroy (OSA_EXT_TaskGetId ());
}

void Job1(osaTaskParam_t argument)
{
  /*Do some work*/
}

void Job2(osaTaskParam_t argument)
{
  if((uint32_t)argument == 1)
  {
    /*Do some work*/
  }
  else
  {
    /*Do some work*/
  }
}
```

3.1.3 Counting Semaphores

3.1.3.1 Overview

The behavior is the same for all operating systems except the allocation procedure. In MQX and MQXLite the semaphore is allocated within the OS Abstraction layer while FreeRTOS allocates it internally.

The osNumberOfSemaphores define controls the maximum number of semaphores permitted.

3.1.3.2 Constant Macro Definitions

Name:

```
#define osNumberOfSemaphores   5       ///< maximum number of semaphores
```

Description:

Defines the maximum number of semaphores.

3.1.3.3 User defined data type definitions

Name:

```
typedef void *osaSemaphoreId_t;
```

Description:

Data type definition for semaphore ID.

3.1.3.4 API Primitives

OSA_EXT_SemaphoreCreate ()

Prototype:

```
osaSemaphoreId_t OSA_EXT_SemaphoreCreate(uint32_t initValue);
```

Description:
Creates and Initializes a Semaphore object used for managing resources.

Parameters:

Name	Type	Direction	Description
initValue	int32_t	[IN]	Initial semaphore count.

Returns:

Semaphore ID for reference by other functions or NULL in case of error.

OSA_EXT_SemaphoreDestroy ()

Prototype:

```
osaStatus_t OSA_EXT_SemaphoreDestroy(osaSemaphoreId_t semId);
```

Description:

Creates and Initializes a Semaphore object used for managing resources.

Parameters:

Name	Type	Direction	Description
semId	osSemaphoreId_t	[IN]	Semaphore ID returned by `OSA_EXT_SemaphoreCreate`.

Returns:

- osaStatus_Success The semaphore is successfully destroyed.
- osaStatus_Error The semaphore can not be destroyed.

OSA_EXT_SemaphoreWait ()

Prototype:

```
osaStatus_t OSA_EXT_SemaphoreWait(osaSemaphoreId_t semId, uint32_t millisec);
```

Description:

Takes a semaphore token.

Parameters:

Name	Type	Direction	Description
semId	osSemaphoreId_t	[IN]	Semaphore ID returned by `OSA_EXT_SemaphoreCreate`.
millisec	uint32_t	[IN]	Timeout value in milliseconds.

Returns:

- osaStatus_Success The semaphore is received.
- osaStatus_Timeout The semaphore is not received within the specified 'timeout'.
- osaStatus_Error An incorrect parameter was passed.

OSA_EXT_SemaphorePost ()

Prototype:

```
osaStatus_t OSA_EXT_SemaphorePost(osaSemaphoreId_t semId);
```

Description:

Releases a semaphore token.

Parameters:

Name	Type	Direction	Description
semId	osSemaphoreId_t	[IN]	Semaphore ID returned by `OSA_EXT_SemaphoreCreate`.

Returns:

- osaStatus_Success The semaphore is successfully signaled.
- osaStatus_Error The object can not be signaled or invalid parameter.

3.1.4 Mutexes

3.1.4.1 Overview

For all operating systems mutexes are implemented with priority inheritance mechanism. In MQX and MQXLite mutexes are much more configurable than in FreeRTOS. The OS Abstraction takes care of the additional configuration steps and sets up the mutex in a way that mimics the FreeRTOS mutex behavior.

3.1.4.2 Constant Macro Definitions

Name:

```
#define osNumberOfMutexes      5       ///< maximum number of mutexes
```

Description:

Defines the maximum number of mutexes.

3.1.4.3 User defined data type definitions

Name:

```
typedef void *osaMutexId_t;
```

Description:

Data type definition for mutex ID.

3.1.4.4 API Primitives

OSA_EXT_MutexCreate ()

Prototype:

```
osaMutexId_t OSA_EXT_MutexCreate(void);
```

Description:
Creates and initializes a mutex.

Parameters:

None

Returns:

Mutex ID for reference by other functions or NULL in case of error.

OSA_EXT_MutexDestroy ()

Prototype:

```
osaStatus_t OSA_EXT_MutexDestroy(osaMutexId_t mutexId);
```

Description:

Destroys the mutex object and free the used memory.

Parameters:

Name	Type	Direction	Description
mutex_id	osMutexId	[IN]	Pointer to the mutex.

Returns:

- osaStatus_Success The mutex is successfully destroyed.
- osaStatus_Error The mutex can not be destroyed.

OSA_EXT_MutexLock ()

Prototype:

```
osaStatus_t OSA_EXT_MutexLock(osaMutexId_t mutexId, uint32_t millisec);
```

Description:
Takes the mutex.

Parameters:

Name	Type	Direction	Description
mutexId	osMutexId	[IN]	Pointer to the mutex.
millisec	uint32_t	[IN]	Number of milliseconds to wait for the mutex to become available.

Returns:

- osaStatus_Success The mutex is locked successfully.
- osaStatus_Timeout Timeout occurred.
- osaStatus_Error Incorrect parameter was passed.

OSA_EXT_MutexUnlock ()

Prototype:

```
osaStatus_t OSA_EXT_MutexUnlock(osaMutexId_t mutexId);
```

Description:
Releases a mutex.

Parameters:

Name	Type	Direction	Description
mutexId	osMutexId	[IN]	ID of the mutex.

Returns:

- osaStatus_Success The mutex is successfully unlocked.
- osaStatus_Error The mutex cannot be unlocked or invalid parameter.

3.1.5 Message queues

3.1.5.1 Overview

The main difference between MQXLite and FreeRTOS regarding message queues is that on MQXLite the user is the one responsible for allocating memory for the queue but in FreeRTOS queue allocation is done by the OS. For this reason queue allocation has been moved in the OS Abstraction layer. In both

operating systems message passing through queues are done by copy and not by reference. Messages are defined to be a single 32 bit value or pointer.

3.1.5.2 Constant Macro Definitions

Name:

```
#define osNumberOfMessageQs    5       ///< maximum number of message queues
```

Description:

Defines the maximum number of message queues.

Name:

```
#define osNumberOfMessages    40       ///< number of messages of all message queues
```

Description:

Defines the total number of messages for all message queues.

3.1.5.3 User defined data type definitions

Name:

```
typedef void* osaMsgQId_t;
```

Description:

Data type definition for message queue ID.

Name:

```
typedef void* osaMsg_t;
```

Description:

Data type definition for queue message type.

3.1.5.4 API Primitives

OSA_EXT_MsgQCreate ()

Prototype:

```
osaMsgQId_t OSA_EXT_MsgQCreate( uint32_t  msgNo);
```

Description:
Creates and initializes a message queue.

Parameters:

Name	Type	Direction	Description
msgNo	Uint32_t	[IN]	Number of messages that the queue should acommodate

Returns:

Message queue handle if successful or NULL if failed.

OSA_EXT_MsgQDestroy ()

Prototype:

```
osaStatus_t OSA_EXT_MsgQDestroy(osaMsgQId_t msgQId);
```

Description:
Destroys a message queue.

Parameters:

Name	Type	Direction	Description
msgQId	osaMsgQId_t	[IN]	The queue hander returned by *OSA_EXT_MsgQCreate()*

Returns:

- osaStatus_Success The queue was successfully destroyed.
- osaStatus_Error Message queue destruction failed.

OSA_EXT_MsgQPut ()

Prototype:

```
osaStatus_t OSA_EXT_MsgQPut(osaMsgQId_t msgQId, osaMsg_t Message);
```

Description:
Puts a message in the message queue.

Parameters:

Name	Type	Direction	Description
msgQId	osaMsgQId_t	[IN]	Message queue ID.
message	osaMsg_t	[IN]	Queue message

Returns:

- osaStatus_Success Message successfully put into the queue.
- osaStatus_Error The queue was full or an invalid parameter was passed.

OSA_EXT_MsgQGet ()

Prototype:

```
osaStatus_t OSA_EXT_MsgQGet(osaMsgQId_t msgQId, osaMsg_t message, uint32_t millisec);
```

Description:
Gets a message from the message queue.

Parameters:

Name	Type	Direction	Description
msgQId	osaMsgQId_t	[IN]	Message queue ID.
message	osaMsg_t	[IN]	Queue message
millisec	uint32_t	[IN]	Time to wait for a message to arrive or *osaWaitForever_c* in case of infinite time.

Returns:

- osaStatus_Success if the message successfully obtained from the queue.
- osaStatus_Timeout if the queue remains empty after timeout.
- osaStatus_Error for invalid parameter.

3.1.6 Events

3.1.6.1 Overview

When waiting for events, a mask is passed to the API which indicates the desired event flags to wait for. If the mask is *osaEventFlagsAll_c*, it shall wait for any signal flag. Else it shall wait for one/all flags to be set.

3.1.6.2 Constant Macro Definitions

Name:

```
#define osNumberOfEvents      5       ///< maximum number of Signal Flags available
```

Description:

The number of event objects

3.1.6.3 User defined data type definitions

Name:

```
typedef void* osaEventId_t;
```

Description:

Data type definition for the event objects; this is a pointer to a pool of objects.

3.1.6.4 API Primitives

OSA_EXT_EventCreate ()

Prototype:

```
osaEventId_t OSA_EXT_EventCreate(bool_t autoClear);
```

Description:
Set the specified signal flags of an active thread.

Parameters:

Name	Type	Direction	Description
autoClear	Bool_t	[IN]	If TRUE, event flags will be automatically cleared. Else, OSA_EXT_EventClear() must be called to clear the flags manually.

Returns:

Event Id if success, NULL if creation failed!

OSA_EXT_EventDestroy ()

Prototype:

```
osaStatus_t OSA_EXT_EventDestroy(osaEventId_t eventId);
```

Description:
Set the specified signal flags of an active thread.

Parameters:

Name	Type	Direction	Description
eventId	osaEventId_t	[IN]	The event Id

Returns:

- osaStatus_Success The event is successfully destroyed.
- osaStatus_Error Event destruction failed.

OSA_EXT_EventSet ()

Prototype:

```
osaStatus_t OSA_EXT_EventSet
(
osaEventId_t eventId,
osaEventFlags_t flagsToSet
);
```

Description:
Set the specified signal flags of an active thread.

Parameters:

Name	Type	Direction	Description
eventId	osaEventId_t	[IN]	The event Id
flagsToClear	osaEventFlags _t	[IN]	flags to set.

Returns:

- osaStatus_Success The flags were successfully set.
- osaStatus_Error An incorrect parameter was passed.

OSA_EXT_EventClear ()

Prototype:

```
osaStatus_t OSA_EXT_EventClear
(
osaEventId_t eventId,
osaEventFlags_t flagsToClear
);
```

Description:
Clear the specified event flags of an active thread.

Parameters:

Name	Type	Direction	Description
eventId	osaEventId_t	[IN]	The event Id
flagsToClear	osaEventFlags _t	[IN]	flags to clear.

Returns:

- osaStatus_Success The flags were successfully cleared.
- osaStatus_Error An incorrect parameter was passed.

OSA_EXT_EventWait ()

Prototype:

```
osaStatus_t OSA_EXT_EventWait
(
osaEventId_t eventId,
osaEventFlags_t flagsToWait,
bool_t waitAll,
uint32_t millisec,
osaEventFlags_t *pSetFlags
);
```

Description:
Waits for one or more event flags to become signaled for the calling thread.

Parameters:

Name	Type	Direction	Description
eventId	osaEventId_t	[IN]	The event Id
flagsToWait	osaEventFlags _t	[IN]	flags to wait for.
waitAll	Bool_t	[IN]	If TRUE, then it will wait for all flags to be set before releasing the task
millisec	uint32_t	[IN]	Time to wait for signal or 0 in case of infinite time.
pSetFlags	osaEventFlags_t*	[OUT]	Pointer to a location where to store the flags that have bees set

Returns:

- osaStatus_Success The wait condition met and function returns successfully.
- osaStatus_Timeout Has not met wait condition within timeout.
- osaStatus_Error An incorrect parameter was passed.

3.1.7 Timers

3.1.7.1 Overview

When waiting for events, a mask is passed to the API which indicates the desired event flags to wait for. If the mask is *osaEventFlagsAll_c*, it shall wait for any signal flag. Else it shall wait for one/all flags to be set.

3.1.7.2 Constant Macro Definitions

Name:

```
#define osNumberOfTimers      1       ///< maximum number of OS timers available
```

Description:

The number of OS timer objects

Name:

```
#define OSA_EXT_TIMER_DEF(name, function)
```

Description:

Defines an OS timer object using name and callback function.

Name:

```
#define OSA_EXT_TIMER(name)
```

Description:

Define used to access an OS timer object.

3.1.7.3 User defined data type definitions

Name:

```
typedef void* osaTimerId_t;
```

Description:

Data type definition for the timer objects; this is a pointer to a pool of objects.

Name:

```
typedef void (*osaTimerFctPtr_t) (void const *argument);
```

Description:

Timer callback function type.

3.1.7.4 API Primitives

OSA_EXT_TimerCreate ()

Prototype:

```
osaTimerId_t OSA_EXT_TimerCreate (osaTimerDef_t *timer_def, osaTimer_t type, void *argument);
```

Description:
Creates and initializes an OS timer object.

Parameters:

Name	Type	Direction	Description
timer_def	osaTimerDef_t	[IN]	Timer object
type	osaTimer_t	[IN]	osaTimer_Once for one-shot or osaTimer_Periodic for periodic behavior
argument	void *	[IN]	Parameter passed to the callback function

Returns:

Event Id if success, NULL if creation failed!

OSA_EXT_TimerStart ()

Prototype:

```
osaStatus_t OSA_EXT_TimerStart (osaTimerId_t timer_id, uint32_t millisec);
```

Description:
Starts or restarts a timer.

Parameters:

Name	Type	Direction	Description
timer_id	osaTimerId_t	[IN]	Timer Id, returned by the OSA_EXT_TimerCreate()
millisec	Uint32_t	[IN]	Delay value for the timer

Returns:

status code that indicates the execution status of the function.

OSA_EXT_TimerStop ()

Prototype:

```
osaStatus_t OSA_EXT_TimerStop (osaTimerId_t timer_id);
```

Description:
Stop a timer.

Parameters:

Name	Type	Direction	Description
timer_id	osaTimerId_t	[IN]	Timer Id, returned by the OSA_EXT_TimerCreate()

Returns:

status code that indicates the execution status of the function.

OSA_EXT_TimerDestroy ()

Prototype:

```
osaStatus_t OSA_EXT_TimerDestroy (osaTimerId_t timer_id);
```

Description:
Dequeues the timer from the os and dealocates it from the timers heap.

Parameters:

Name	Type	Direction	Description
timer_id	osaTimerId_t	[IN]	Timer Id, returned by the OSA_EXT_TimerCreate()

Returns:

status code that indicates the execution status of the function.

3.1.8 Interrupts

3.1.8.1 API Primitives

OSA_EXT_InterruptDisable ()

Prototype:

```
void OSA_EXT_InterruptDisable(void);
```

Description:
Disables all interrupts.

Parameters:

None.

Returns:

None.

OSA_EXT_InterruptEnable ()

Prototype:

```
void OSA_EXT_InterruptEnable (void);
```

Description:
Enables all interrupts.

Parameters:

None.

Returns:

None.

OSA_EXT_InstallIntHandler ()

Prototype:

```
void* OSA_EXT_InstallIntHandler(uint32_t IRQNumber, void (*handler)(void));
```

Description:
Installs and ISR for the specified IRQ.

Parameters:

Name	Type	Direction	Description
IRQNunber	uint32_t	[IN]	IRQ number
handler	void (*handler)(void)	[IN]	Handler function

Returns:

- If successful, returns previous ISR handler
- Retuens NULL if the handler could not be installed.

3.2 Message Management

3.2.1 Overview

Included in the framework, there is a memory management module which is organized in partitions of identical memory blocks. Every block of memory has a header used by the memory manager for internal book keeping. This header is reused in the message system to avoid further overhead. The message component can only use buffers allocated with the memory manager due to this behavior.

The framework also includes a general purpose linked lists module which is used in several other components. The message system takes advantage of linked lists, using the memory manager's header space, to provide an overhead-free unlimited size queue system. The user shall allocate a message buffer and then pass it to the message system without worrying about the extra space required by the linked list element header or the maximum size of the queue. The only limitation is the amount of memory available to the memory manager.

Although this approach is efficient in terms of memory space and unbound by a maximum queue size, it does not provide the means to synchronize tasks. For this purpose the user can turn to semaphores, mutexes, signals etc. It makes sense to use the existing signals, since they are created for every task, to avoid consuming extra memory.

Now the user can send a message to a receiving task and then activate the synchronization element, and on the other side it shall first wait for the synchronization signal and then dequeue the message. The actual memory buffer is allocated by the sending task and must be freed or reused by the receiving task. Using this approach only requires an additional linked list anchor in terms of memory space.

The messaging module is a blend of macros and functions on top of the Memory Manager and Lists modules.

3.2.2 Data type definitions

Name:

```
#define  anchor_t        list_t
#define  msgQueue_t      list_t
```

Description:

Defines the anchor and message queue type definition. See the Lists module for more information.

3.2.3 Message Management API Primitives

3.2.3.1 MSG_Queue

Prototype:

```
#define MSG_Queue(anchor, element) ListAddTailMsg((anchor), (element))
```

Description:
Puts a message in the message queue.

Parameters:

Name	Type	Direction	Description
anchor	msgQueue_t	[IN]	Pointer to the message queue.
element	void *	[IN]	Buffer allocated with the Memory manager.

Returns:

None.

3.2.3.2 MSG_DeQueue

Prototype:

```
#define MSG_DeQueue(anchor) ListRemoveHeadMsg(anchor)
```

Description:
Dequeues a message from the message queue.

Parameters:

Name	Type	Direction	Description
anchor	msgQueue_t	[IN]	Pointer to the message queue.

Returns:

Pointer to the message buffer.

3.2.3.3 MSG_Pending

Prototype:

```
#define MSG_Pending(anchor) ((anchor)->head != 0)
```

Description:
Check if a message is pending in a queue.

Parameters:

Name	Type	Direction	Description
anchor	msgQueue_t	[IN]	Pointer to the message queue.

Returns:

Returns TRUE if any pending messages, and FALSE otherwise.

3.2.3.4 MSG_InitQueue

Prototype:

```
#define MSG_InitQueue(anchor) ListInitMsg(anchor)
```

Description:
Initializes a message queue.

Parameters:

Name	Type	Direction	Description
anchor	msgQueue_t	[IN]	Pointer to the message queue.

Returns:

None.

3.2.3.5 List_ClearAnchor

Prototype:

```
#define List_ClearAnchor(anchor) ListInitMsg(anchor)
```

Description:
Resets a message queue.

Parameters:

Name	Type	Direction	Description
Anchor	msgQueue_t	[IN]	Pointer to the message queue.

Returns:

None.

3.2.3.6 MSG_Alloc

Prototype:

```
#define  MSG_Alloc(element)          MEM_BufferAlloc(element)
```

Description:
Allocates a message.

Parameters:

Name	Type	Direction	Description
element	uint32_t	[IN]	Size of the buffer to be allocated with the Memory manager.

Returns:

Pointer to the newly allocated block or NULL if failed.

3.2.3.7 MSG_AllocType

Prototype:

```
#define  MSG_AllocType(type)          MEM_BufferAlloc(sizeof(type))
```

Description:
Allocates a data type.

Parameters:

Name	Type	Direction	Description
type	-	[IN]	Data type to be allocated.

Returns:

Pointer to the newly allocated block or NULL if failed.

3.2.3.8 MSG_Free

Prototype:

```
#define  MSG_Free(element)           MEM_BufferFree(element)
```

Description:
Frees a message.

Parameters:

Name	Type	Direction	Description
element	void *	[IN]	Pointer to buffer to free.

Returns:

Status of the free operation.

3.2.3.9 ListInitMsg

Prototype:

```
#define ListInitMsg(listPtr) ListInit((listPtr), 0)
```

Description:
Initializes a list.

Parameters:

Name	Type	Direction	Description
listPtr	listHandle_t	[IN]	Pointer to the list.

Returns:

None.

3.2.3.10 ListAddTailMsg

Prototype:

```
void ListAddTailMsg ( listHandle_t list, void* buffer );
```

Description:
Adds a buffer to the end of the list.

Parameters:

Name	Type	Direction	Description
list	listHandle_t	[IN]	Pointer to the list.
buffer	void *	[IN]	Pointer to the buffer.

Returns:

None.

3.2.3.11 ListRemoveHeadMsg

Prototype:

```
void *ListRemoveHeadMsg( listHandle_t list );
```

Description:
Removes a buffer from the head of the list.

Parameters:

Name	Type	Direction	Description
list	listHandle_t	[IN]	Pointer to the list.

Returns:

Pointer to the removed buffer.

3.2.3.12 ListGetHeadMsg

Prototype:

```
void *ListGetHeadMsg   ( listHandle_t list );
```

Description:
Gets a buffer from the head of the list, without removing it.

Parameters:

Name	Type	Direction	Description
list	listHandle_t	[IN]	Pointer to the list.

Returns:

Pointer to the head buffer.

3.2.3.13 ListGetNextMsg

Prototype:

```
void *ListGetNextMsg   ( void* buffer );
```

Description:
Gets the next linked buffer.

Parameters:

Name	Type	Direction	Description
buffer	void *	[IN]	Pointer to a list element.

Returns:

Pointer to the next buffer.

3.3 Memory Management

3.3.1 Overview

A generic allocation scheme is not desirable to be used due to the memory fragmentation issues that usually appear and therefore the framework must provide a non-fragmenting memory allocation solution. The solution relies on partitions to avoid the memory fragmentation problem and also, the execution time is deterministic. Each partition has a fixed number of partition blocks and each block has a fixed size. The memory management services are implemented using multiple partitions of different sizes. All partitions use memory from a single global array. When a new buffer is requested to be allocated, the framework will return the first available partition block of equal or higher size (i.e. if no buffer of the requested size is available, the allocation routine will return a buffer of a larger size). As requirements, the partition shall be defined in ascending size order and the block sizes must be multiples of 4 to ensure block alignment to 4 bytes:

/*Defines pools by block size and number of blocks. Must be aligned to 4 bytes.*/

```
#ifndef PoolsDetails_c
#define PoolsDetails_c \
         _block_size_  64  _number_of_blocks_    8 _eol_  \
         _block_size_ 128  _number_of_blocks_    4 _eol_  \
         _block_size_ 256  _number_of_blocks_    6 _eol_
#endif
```

For this example, there are three partitions that need to be created. In addition to the requested amount of memory, each block will use a header for internal book keeping. This header should not be used by the user since some information is still needed for deallocation.

3.3.2 Data type definitions

Name:

```
typedef enum
{
MEM_SUCCESS_c = 0,
MEM_INIT_ERROR_c,
MEM_ALLOC_ERROR_c,
MEM_FREE_ERROR_c,
MEM_UNKNOWN_ERROR_c
}memStatus_t;
```

Description:

Defines statuses used in MEM_BufferAlloc and MEM_BufferFree.

3.3.3 Memory Management API Primitives

3.3.3.1 MEM_Init()

Prototype:

```
memStatus_t MEM_Init
(
void
);
```

Description:

This function is used to initialize the memory management sub-system. The function allocates memory for all partitions; initializes the partitions and partition blocks. The function must be called before any other memory management API function.

Parameters:

None.

Returns:

The function returns `MEM_SUCCESS_c` if initialization succeeds or `MEM_INIT_ERROR_c` otherwise.

3.3.3.2 MEM_BufferAlloc()

Prototype:

```
void* MEM_BufferAlloc
(
    uint32_t numBytes
);
```

Description:

The function allocates a buffer from an existing partition with free blocks. The size of the allocated buffer is equal or greater with the requested one.

Parameters:

Name	Type	Direction	Description
numBytes	uint32_t	[IN]	Requested buffer size, in bytes

Returns:

A pointer to the allocated buffer (partition block) or NULL if the allocation operation fails.

3.3.3.3 MEM_BufferFree()

Prototype:

```
memStatus_t MEM_BufferFree
(
    void* buffer
);
```

Description:

The function attempts to free the buffer passed as function argument.

Parameters:

Name	Type	Direction	Description
buffer	void *	[IN]	The buffer to be freed

Returns:

If the buffer is successfully freed, the function returns `MEM_SUCCESS_c`, otherwise it returns `MEM_FREE_ERROR_c`.

NOTE

User must call MEM_BufferFree() only on a pointer thet was returned by MEM_BufferAlloc().

Also one must not call the free function for a buffer that is already free!

3.3.3.4 MEM_GetAvailableBlocks()

Prototype:

```
uint32_t MEM_GetAvailableBlocks
(
    uint32_t size
);
```

Description:

The function queries how many buffers with a size equal or greater than the specified one are available.

Parameters:

Name	Type	Direction	Description
size	uint32_t	[IN]	The size on which basis the buffers are queried

Returns:

The buffers count that satisfied the condition to have their size equal or greater with the one specified by the function argument.

3.3.3.5 MEM_BufferGetSize()

Prototype:

```
uint16_t MEM_BufferGetSize(void* buffer);
```

Description:

The function queries the size of the given buffer.

Parameters:

Name	Type	Direction	Description
buffer	void *	[IN]	Pointer to the allocated buffer.

Returns:

The buffer size.

3.3.3.6 MEM_WriteReadTest()

Prototype:

```
uint32_t MEM_WriteReadTest
(
    void
);
```

Description:

The function performs a write-read-verify test across all pools

Parameters:

None.

Returns:

Returns `MEM_SUCCESS_c` if test was successful, `MEM_ALLOC_ERROR_c` if a buffer was not allocated successfuly, `MEM_FREE_ERROR_c` if a buffer was not freed successfuly or `MEM_UNKNOWN_ERROR_c` if a verify error, heap overflow or data corruption occurred.

3.3.4 Sample Code

Memory allocation

```
uint8_t * buffer = MEM_BufferAlloc(64);
if(NULL == buffer)
{
    ... error ...
}

...

/* Free buffer */
if(MEM_SUCCESS_c != MEM_BufferFree(buffer))
{
    ... error ...
}

...


/* Check availabe blocks */
if(MEM_GetAvailableBlocks(128) < 3)
{
        ... error ...
}
```

3.4 Timers Manager

3.4.1 Overview

The Timers Manager offers timing services with increased resolution as compared to the OS based timing functions. The Timers Manager operates at the peripheral clock frequency, while the OS timer frequency is set to 200Hz (5 ms period).

The following services are provided by the Timers Manager:

- Module initialization
- Allocate a timer
- Free a timer
- Enable a timer
- Start a timer
- Stop a timer
- Check if a timer is active
- Obtain the remaining time until a specified timer timeouts

There are two types of timers provided:

Single Shot Timers – will run only once until timeout. They can be stopped before the timeout.

Interval Timers – will run continuously and timeout at the set regular intervals until explicitly stopped.

Each allocated timer has an associated callback function that is called from interrupt execution context, and therefore must not call blocking OS APIs. They can have the potential to block, but that potential must never materialize.

Note

The exact time at which a callback is executed will be actually greater or equal than the requested value.

The timer resolution is 1 ms but it is recommended to use a multiple of 4 ms as timeout values to increase accuracy. This is due to internal integer calculation errors.

The implementation of the Timers Manager on Kinetis MCU based platforms uses either FTM or TPM peripheral. An interrupt is generated each time an allocated and running timer timeouts, so the mechanism is more efficient as compared to the OS managed timing, which requires the execution of periodic (default 5ms) interrupts.

Timers can be identified as low power timers on creation. Usually, this means that low power timers will not run in low power modes rather they will be synchronized when exiting from low power. If a low power timer expires when the MCU sleeps its expiration will be processed when the MCU exits sleep.

If HW low power timers are enabled (*gTMR_EnableHWLowPowerTimers_d = 1*), then MCU will exit from sleep when the nearest one to expire will!

The Timers Manager creates a task to handle the internal processing required. All callbacks are called in the context and with the priority of the timer task. As a general guideline callbacks must be non-blocking and short. They shouldn't do much besides issuing a synchronization signal. The task is set up with an above normal priority.

The Timers Manager module also provides timestamp functionality. This is implemented on top of the RTC and PIT. The RTC peripheral is running in all low power modes. In addition, there is also the possibility to set the absolute time with a 30 microseconds resolution and register an alarm event in absolute or relative time with a 1 second resolution. Please note the fact that there may be other framework components that use alarms and only one registered alarm event at a time is permitted. The RTC section of the timers module requires its own initialization.

3.4.2 Constant Macro Definitions

Name:

```
#define gTMR_Enabled_d                    TRUE
```

Description:

Enables/Disabled the timer module except RTC functionality.

Name:

```
#define gTimestamp_Enabled_d              TRUE
```

Description:

Enables / disables the timestamp functionality.

Name:

```
#define gTMR_PIT_Timestamp_Enabled_d              FALSE
```

Description:

The default HW used for timestamp is the RTC. If this define is set to TRUE, then the PIT HW is used for timestamp.

Name:

```
#define gTMR_EnableLowPowerTimers_d       TRUE
```

Description:

Enables / disables the timer synchronization after an exit from low power.

Name:

```
#define gTMR_EnableHWLowPowerTimers_d       FALSE
```

Description:

If set to TRUE, all timers of type *gTmrLowPowerTimer_c*, will use the LPTMR HW. These timers will also run in low power mode!

Name:

```
#define gTMR_EnableMinutesSecondsTimers_d  TRUE
```

Description:

Enables/Disables the `TMR_StartMinuteTimer` and `TMR_StartSecondTimer` wrapper functions.

Name:

```
#define gTmrApplicationTimers_c           0
```

Description:

Defines the number of software timers that can to be used by the application.

Name:

```
#define gTmrStackTimers_c                 1
```

Description:

Defines the number of stack timers that can to be used by the stack.

Name:

```
#define gTmrTaskPriority_c                2
```

Description:

Defines the priority of the timer task.

Name:

```
#define gTmrTaskStackSize_c               500
```

Description:

Defines the stack size (in bytes) of the timer task.

Name:

```
#define TmrSecondsToMicroseconds( n )     ( (uint64_t) (n * 1000000) )
```

Description:

Converts seconds to microseconds.

Name:

```
#define TmrMicrosecondsToSeconds( n )     ( n / 1000000 )
```

Description:

Converts microseconds to seconds.

Name:

```
#define gTmrInvalidTimerID_c    0xFF
```

Description:

Reserved value for invalid timer ID.

Name:

```
#define gTmrSingleShotTimer_c   0x01
#define gTmrIntervalTimer_c     0x02
#define gTmrSetMinuteTimer_c    0x04
#define gTmrSetSecondTimer_c    0x08
#define gTmrLowPowerTimer_c     0x10
```

Description:

Timer types coded values.

Name:

```
#define gTmrMinuteTimer_c       ( gTmrSetMinuteTimer_c )
```

Description:

Minute timer definition.

Name:

```
#define gTmrSecondTimer_c       ( gTmrSetSecondTimer_c )
```

Description:

Second timer definition.

Name:

```
#define gTmrLowPowerMinuteTimer_c          ( gTmrMinuteTimer_c | gTmrLowPowerTimer_c )
#define gTmrLowPowerSecondTimer_c          ( gTmrSecondTimer_c | gTmrLowPowerTimer_c )
#define gTmrLowPowerSingleShotMillisTimer_c ( gTmrSingleShotTimer_c | gTmrLowPowerTimer_c )
#define gTmrLowPowerIntervalMillisTimer_c   ( gTmrIntervalTimer_c | gTmrLowPowerTimer_c )
```

Description:

Low power/minute/second/millisecond timer definitions.

3.4.3 User defined data type definitions

Name:

```
typedef uint8_t     tmrTimerID_t;
```

Description:

Timer identification data type definition.

Name:

```
typedef uint8_t     tmrTimerType_t;
```

Description:

Timer type data definition.

Name:

```
typedef uint16_t tmrTimerTicks16_t;
typedef uint32_t tmrTimerTicks32_t;
typedef uint64_t tmrTimerTicks64_t;
```

Description:

16, 32 and 64 bit timer ticks type definition.

Name:

```
typedef void ( *pfTmrCallBack_t ) ( void * );
```

Description:

Callback pointer definition.

Name:

```
typedef uint32_t    tmrTimeInMilliseconds_t;
typedef uint32_t    tmrTimeInMinutes_t;
typedef uint32_t    tmrTimeInSeconds_t;
```

Description:

Time specified in milliseconds, minutes and seconds.

Name:

```
typedef enum tmrErrCode_tag{
    gTmrSuccess_c,
    gTmrInvalidId_c,
    gTmrOutOfRange_c
}tmrErrCode_t;
```

Description:

The error code returned by all TMR_Start… functions

3.4.4 System Timer API Primitives

3.4.4.1 TMR_Init()

Prototype:

```
void TMR_Init
(
    void
);
```

Description:

The function initializes the system timer module and must be called before the module is used. Internally, the function creates the timer task, calls the low level driver initialization function, configures and starts the hardware timer and initializes module internal variables.

Parameters:

None

Returns:

None

3.4.4.2 TMR_AllocateTimer()

Prototype:

```
tmrTimerID_t TMR_AllocateTimer
(
    void
);
```

Description:

This function is used to allocate a timer. Before staring or stopping a timer, it must be first allocated. After the timer is allocated, its internal status is set to inactive.

Parameters:

None

Returns:

The function return the allocated timer ID or *gTmrInvalidTimerID* if no timers are available. The returned timer ID has to be used by the application for all further interactions with the allocated timer, until the timer is freed.

3.4.4.3 TMR_FreeTimer()

Prototype:

```
void TMR_FreeTimer
(
    tmrTimerID_t timerID
);
```

Description:

The function frees the specified timer if the application no longer needs it.

Parameters:

Name	Type	Direction	Description
timerID	tmrTimerID_t	[IN]	The ID of the timer to be freed

Returns:

None

3.4.4.4 TMR_GetMaxTimeMs ()

Prototype:

```
uint32_t TMR_GetMaxTimeMs
(
    void
);
```

Description:

The function determines the maximum number of miliseconds that can be programmed.

Parameters:

None

Returns:

Maximum miliseconds

3.4.4.5 TMR_GetMaxLpTimeMs ()

Prototype:

```
uint32_t TMR_GetMaxLpTimeMs
(
    void
);
```

Description:

The function determines the maximum number of miliseconds that can be programmed into the hardware low power timer module.

Parameters:

None

Returns:

Maximum miliseconds

3.4.4.6 TMR_StartTimer()

Prototype:

```
tmrErrCode_t TMR_StartTimer
(
    tmrTimerID_t timerID,
    tmrTimerType_t timerType,
    tmrTimeInMilliseconds_t timeInMilliseconds,
    void (*pfTimerCallBack)(void *),
    void *param
);
```

Description:

The function is used by the application to setup and start a (pre-) allocated timer. If the specified timer is already running, calling this function will stop the timer and reconfigure it with the new parameters.

Parameters:

Name	Type	Direction	Description
timerID	tmrTimerID_t	[IN]	The ID of the timer to be started
timerType	tmrTimerType_t	[IN]	The type of the timer to be started
timeInMilliseconds	tmrTimeInMilliseconds_t	[IN]	Timer counting interval

Name	Type	Direction	Description
			expressed in system ticks
pfTimerCallBack	void (*)(void *)	[IN]	Pointer to the callback function
param	void *	[IN]	Parameter to be passed to the callback function.

Returns:

The error code

3.4.4.7 TMR_StopTimer()

Prototype:

```
void TMR_StopTimer
(
    tmrTimerID_t timerID
);
```

Description:

The function is used by the application to stop a pre-allocated running timer. If the specified timer is already stopped, calling this function is harmless. Stopping a timer will not automatically free it. After it is stopped, the specified timer timeout events will be deactivated until the timer is re-started.

Parameters:

Name	Type	Direction	Description
timerID	tmrTimerID_t	[IN]	The ID of the timer to be stopped

Returns:

None

3.4.4.8 TMR_IsTimerActive

Prototype:

```
bool_t TMR_IsTimerActive
(
    tmrTimerID_t timerID
);
```

Description:

The function checks if the specified timer is active.

Parameters:

Name	Type	Direction	Description
timerID	tmrTimerID_t	[IN]	The ID of the timer to be checked

Returns:

TRUE if the timer is active, FALSE otherwise

3.4.4.9 TMR_GetRemainingTime

Prototype:

```
uint32_t TMR_GetRemainingTime
(
    tmrTimerID_t tmrID
);
```

Description:

The function returns the time (expressed in milliseconds) until the specified timer expires (timeouts) or zero if the timer is inactive or already expired.

Parameters:

Name	Type	Direction	Description
tmrID	tmrTimerID_t	[IN]	The ID of the timer

Returns:

See description

3.4.4.10 TMR_EnableTimer

Prototype:

```
void TMR_EnableTimer
(
    tmrTimerID_t tmrID
);
```

Description:

The function is used to enable the specified timer.

Parameters:

Name	Type	Direction	Description
tmrID	tmrTimerID_t	[IN]	The ID of the timer to be enabled

Returns:

None

3.4.4.11 TMR_AreAllTimersOff

Prototype:

```
bool_t TMR_AreAllTimersOff
(
    void
);
```

Description:

Check if all timers except the low power timers are OFF.

Parameters:

None

Returns:

TRUE if there are no active non-low power timers, FALSE otherwise.

3.4.4.12 TMR_IsTimerReady

Prototype:

```
bool_t TMR_IsTimerReady
(
    tmrTimerID_t timerID
);
```

Description:

Check if a specified timer is ready.

Parameters:

Name	Type	Direction	Description
timerID	tmrTimerID_t	[IN]	The ID of the timer to be enabled

Returns:

TRUE if the timer (specified by the timerID) is ready, FALSE otherwise.

3.4.4.13 TMR_StartLowPowerTimer

Prototype:

```
tmrErrCode_t TMR_StartLowPowerTimer
(
    tmrTimerID_t timerId,
    tmrTimerType_t timerType,
    uint32_t timeIn,
    void (*pfTmrCallBack)(void *),
    void *param
);
```

Description:

Start a low power timer. When the timer goes off, call the callback function in non-interrupt context. If the timer is running when this function is called, it will be stopped and restarted.

Start the timer with the following timer types:

- gTmrLowPowerMinuteTimer_c
- gTmrLowPowerSecondTimer_c
- gTmrLowPowerSingleShotMillisTimer_c
- gTmrLowPowerIntervalMillisTimer_c

The MCU can enter in low power if there are only active low power timers.

Parameters:

Name	Type	Direction	Description
timerID	tmrTimerID_t	[IN]	The ID of the timer
timerType	tmrTimerType_t	[IN]	The type of the timer
timeIn	uint32_t	[IN]	Time in ticks
pfTimerCallBack	void (*)(void *)	[IN]	Pointer to the callback function
param	void *	[IN]	Parameter to be passed to the callback function.

Returns:

The error code.

3.4.4.14 TMR_StartMinuteTimer

Prototype:

```
tmrErrCode_t TMR_StartMinuteTimer
(
    tmrTimerID_t timerId,
    tmrTimeInMinutes_t timeInMinutes,
    void (*pfTmrCallBack)(void *),
    void *param
);
```

Description:

Start a minute timer.

Parameters:

Name	Type	Direction	Description
timerID	tmrTimerID_t	[IN]	The ID of the timer
timeInMinutes	tmrTimeInMinutes_t	[IN]	Time in minutes
pfTmrCallBack	void (*)(void *)	[IN]	Pointer to the callback function
param	void *	[IN]	Parameter to be passed to the callback function.

Returns:

The error code.

3.4.4.15 TMR_StartSecondTimer

Prototype:

```
tmrErrCode_t TMR_StartMinuteTimer
(
    tmrTimerID_t timerId,
    tmrTimeInSeconds_t timeInSeconds,
    void (*pfTmrCallBack)(void *),
    void *param
);
```

Description:

Start a second timer.

Parameters:

Name	Type	Direction	Description
timerID	tmrTimerID_t	[IN]	The ID of the timer
timeInSeconds	tmrTimeInSeconds_t	[IN]	Time in seconds
pfTmrCallBack	void (*)(void *)	[IN]	Pointer to the callback function
param	void *	[IN]	Parameter to be passed to the callback function.

Returns:

The error code.

3.4.4.16 TMR_StartIntervalTimer

Prototype:

```
tmrErrCode_t TMR_StartIntervalTimer
(
    tmrTimerID_t timerId,
    tmrTimeInMilliseconds_t timeInMilliseconds,
    void (*pfTmrCallBack)(void *),
    void *param
);
```

Description:

Start an interval timer.

Parameters:

Name	Type	Direction	Description
timerID	tmrTimerID_t	[IN]	The ID of the timer
timeInMilliseconds	tmrTimeInMilliseconds_t	[IN]	Time in milliseconds
pfTmrCallBack	void (*)(void *)	[IN]	Pointer to the callback function
param	void *	[IN]	Parameter to be passed to the callback function.

Returns:

The error code.

3.4.4.17 TMR_StartSingleShotTimer

Prototype:

```
tmrErrCode_t TMR_StartIntervalTimer
(
    tmrTimerID_t timerId,
    tmrTimeInMilliseconds_t timeInMilliseconds,
    void (*pfTmrCallBack)(void *),
    void *param
);
```

Description:

Start a single shot timer.

Parameters:

Name	Type	Direction	Description
timerID	tmrTimerID_t	[IN]	The ID of the timer
timeInMilliseconds	tmrTimeInMilliseconds_t	[IN]	Time in milliseconds
pfTmrCallBack	void (*)(void *)	[IN]	Pointer to the callback function
param	void *	[IN]	Parameter to be passed to the callback function.

Returns:

The error code.

3.4.4.18 TMR_TimestampInit()

Prototype:

```
void TMR_TimeStampInit
(
    void
);
```

Description:

The function initializes the RTC or PIT HW to enable the timestamp functionallity.

Parameters:

None

Returns:

None

3.4.4.19 TMR_GetTimestamp()

Prototype:

```
uint64_t TMR_GetTimestamp
(
    Void
);
```

Description:

Returns the absolute time at the moment of the call.

Parameters:

None

Returns:

Timestamp in [us]

3.4.4.20 TMR_RTCInit()

Prototype:

```
void TMR_RTCInit
(
    void
);
```

Description:

The function initializes the RTC HW.

Parameters:

None

Returns:

None

3.4.4.21 TMR_RTCGetTimestamp()

Prototype:

```
uint64_t TMR_RTCGetTimestamp
(
    Void
);
```

Description:

Returns the absolute time at the moment of the call, using the RTC.

Parameters:

None

Returns:

Timestamp in [us]

3.4.4.22 TMR_RTCSetTime

Prototype:

```
void TMR_RTCSetTime
(
    uint64_t microseconds
);
```

Description:

Sets the absolute time.

Parameters:

Name	Type	Direction	Description
microseconds	uint64_t	[IN]	Time in microseconds.

Returns:

None.

3.4.4.23 TMR_RTCSetAlarm

Prototype:

```
void TMR_RTCSetAlarm
(
    uint64_t seconds,
    pfTmrCallBack_t callback,
    void *param
);
```

Description:

Sets the alarm in absolute time.

Parameters:

Name	Type	Direction	Description
seconds	uint64_t	[IN]	Time in seconds.
callback	pfTmrCallBack_t	[IN]	Pointer to the callback function
param	void *	[IN]	Parameter to be passed to the callback function.

Returns:

None.

3.4.4.24 TMR_RTCSetAlarmRelative

Prototype:

```
void TMR_RTCSetAlarmRelative
(
    uint32_t seconds,
    pfTmrCallBack_t callback,
    void *param
);
```

Description:

Sets the alarm in relative time.

Parameters:

Name	Type	Direction	Description
seconds	uint32_t	[IN]	Time in seconds.
callback	pfTmrCallBack_t	[IN]	Pointer to the callback function
param	void *	[IN]	Parameter to be passed to the callback function.

Returns:

None.

3.5 Flash Management

3.5.1 Overview

In a standard Harvard architecture based MCU, the flash memory is used to store program code and program constant data. Modern processors have a built-in flash memory controller that can be used under user program execution to store any kind of non-volatile data. Flash memories have individually erasable segments (i.e. sectors) and each segment has a limited number of erase cycles. If the same segments are used all the time to store different kind of data, those segments will become unreliable in short time. Therefore, a wear-leveling mechanism is necessary in order to prolong the service life of the memory. The framework provides a wear-leveling mechanism described in the following paragraphs. The program and erase memory operations are handled by the NVM_Task().

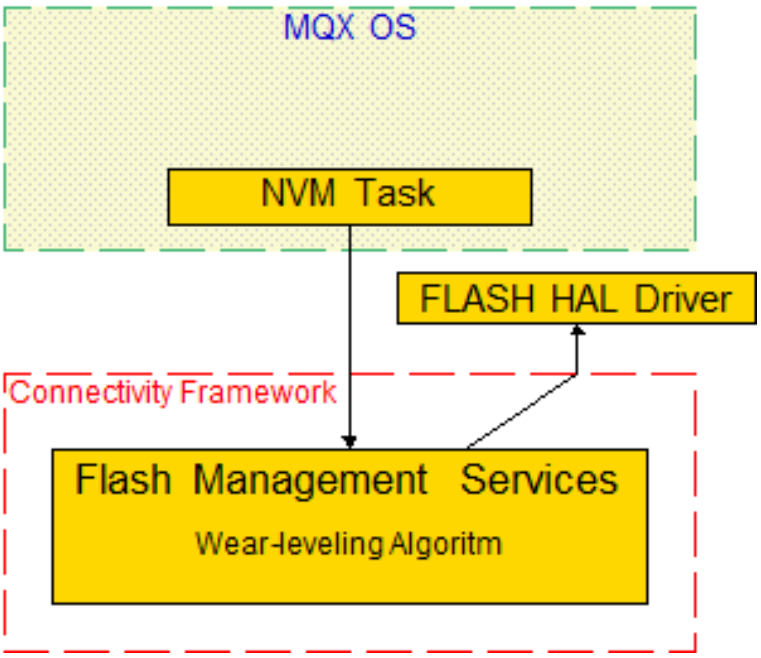

Figure 2. NVM overview

3.5.2 Standard storage system

Most of the MCUs are having only standard FLASH memory that can be used by the Non-volatile storage system. The amount of memory that the system will use for permanent storage is defined in the linker configuration file, as well as its boundaries.

The reserved memory is divided into two pages, called virtual pages. The virtual pages are equally sized and each page is using one or more physical FLASH sectors. Therefore, the smallest configuration is using two physical sectors (one sector per virtual page).

The Flash Management module holds a pointer to a RAM table where the upper layers registers information about the data that should be saved and restored by the storage system. A table entry contains a generic pointer to a contiguous RAM data structure, how many elements the structure is containing, the size of a single element and a table entry ID.

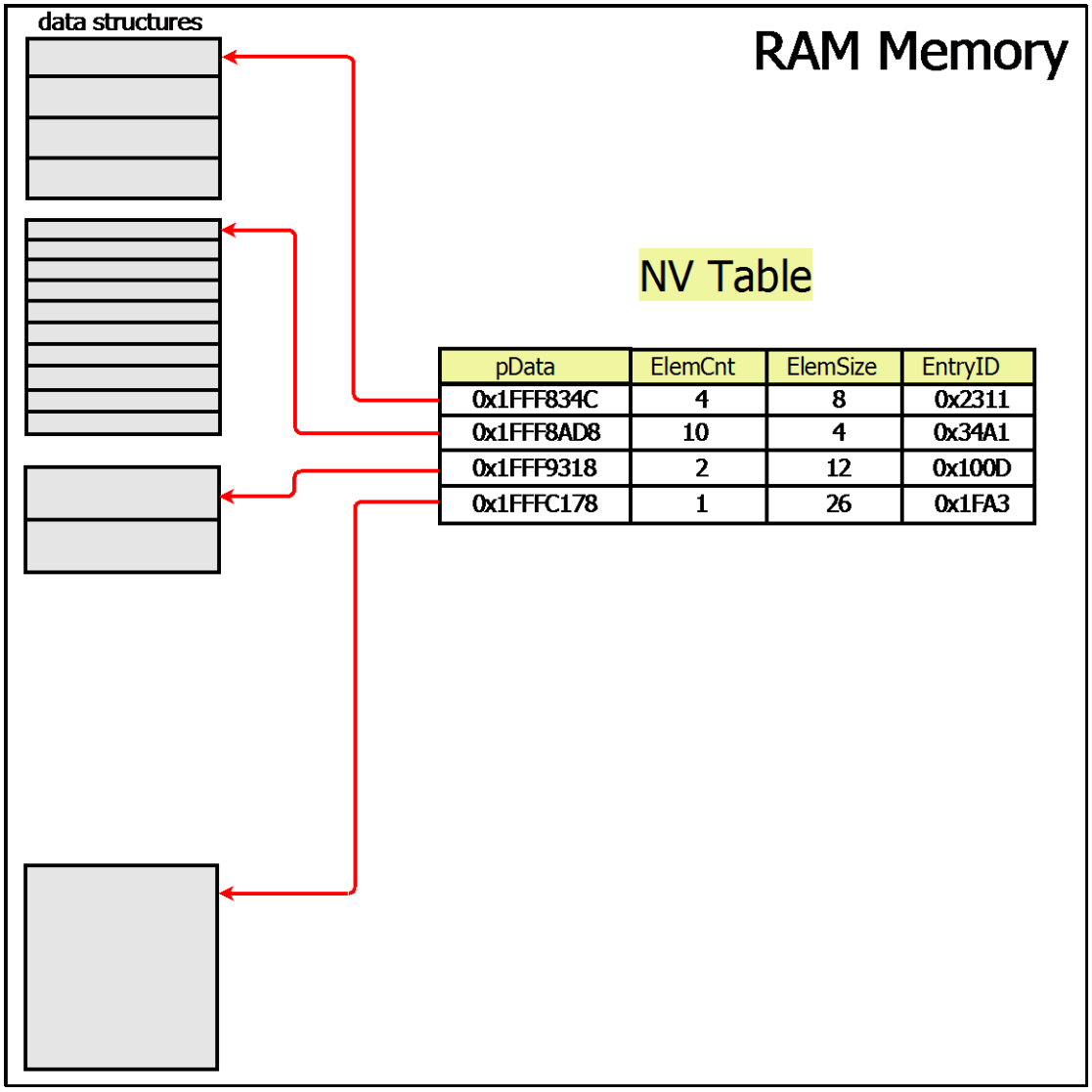

Figure 3. NV Table example

As already presented above, a RAM table entry has the following structure:

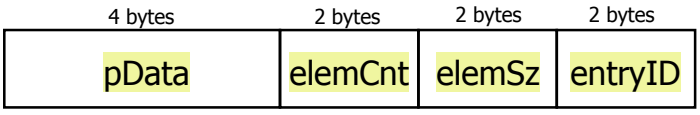

Figure 4. The structure of a NV table entry

Where:

- **pData** is a pointer to a RAM memory location where the dataset elements are stored
- **elemCnt** represents how many elements the dataset has
- **elemSz** is the size of a single element

- **entryID** is a 16-bit unique ID of the dataset

When the data pointed by the table entry pointer (pData) has changed (entirely or just a single element), the upper layers may call the appropriate API function that will make a request to the storage system to save the modified data. All the save operations (except the synchronous save and atomic save), as well as page erase and page copy operations are performed on system idle task. The save is done in one virtual page - the active page. The active page contains information about the records that it holds as well as the records. The storage system may save individual elements of a table entry or the entire table entry.

The page validity is guaranteed by the page counter. The page counter is a 32-bit value and is written at the beginning and at the end of the active page. The values need to be equal to consider the page a valid one. The value of the page counter is incremented after each page copy operation. A page erase operation is performed if the system is formatted or every time a page is full and a new record cannot be written in that page. Before being erased, the page that is full is first copied (only the most recent saves) and afterwards is erased.

The validity of a Meta Information Tag (and therefore of a record) is guaranteed by the MIT start and stop validation bytes. These two bytes needs to be equal to consider the record referred by the MIT as being valid. Furthermore, the value of these bytes indicates the type of the record: single element or entire table entry. The non-volatile storage system allows dynamic changes of the table within the RAM memory as follows:

- remove table entry
- register table entry
- modify the table entry elements count

A new table entry may be successfully registered if there is at least one entry previously removed or if the NV table contains invalid table entries declared explicitly to register new table entries at runtime. The layout of an active page is depicted below:

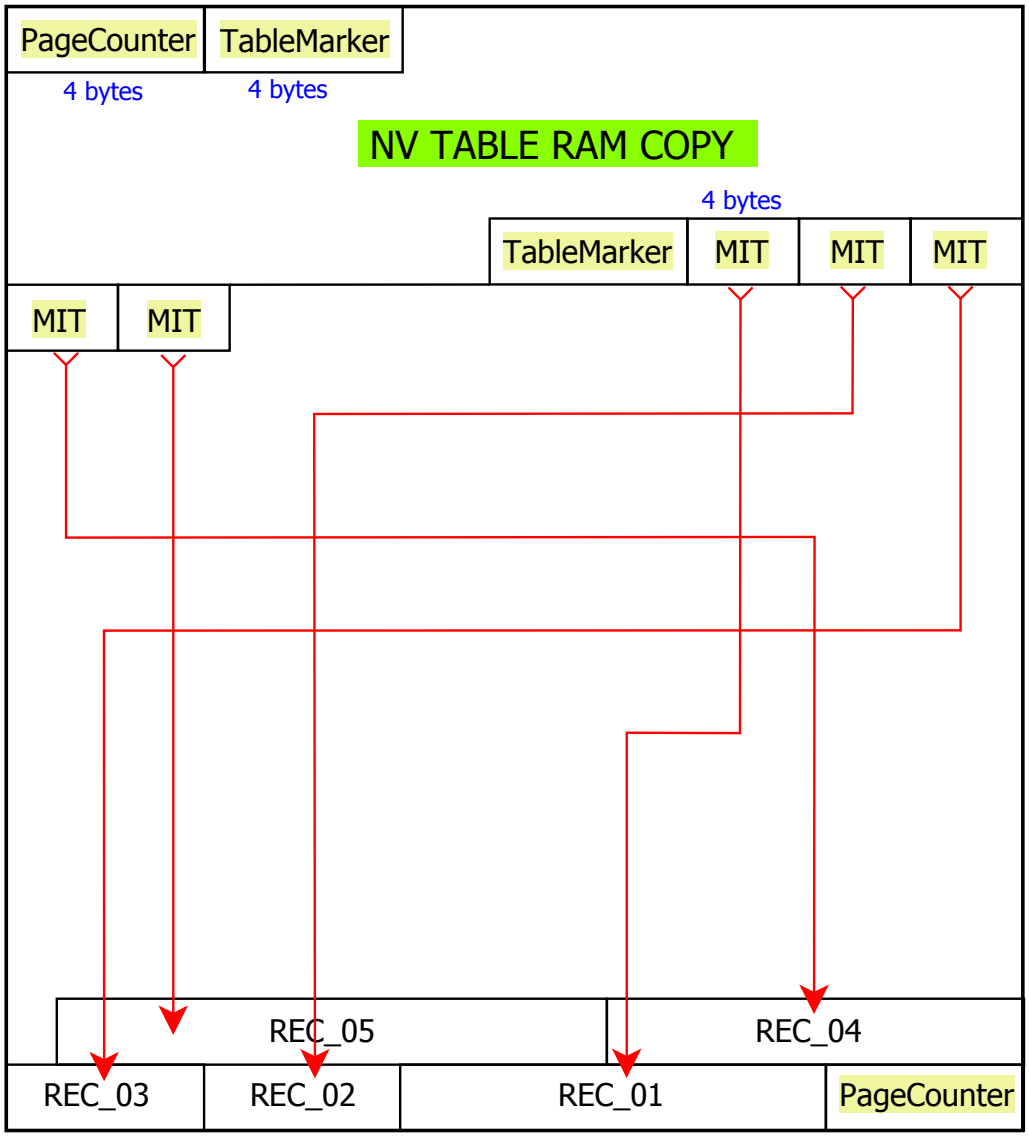

Figure 5. The layout of FLASH active page

As can be observed, the table stored in RAM memory is copied in Flash active page, just after the page counter. The “table start” and “table end” are marked by the so called table markers. The data pointers from RAM are not copied, just the elements count, element size and entry ID. A Flash copy of a RAM table entry has the following structure:

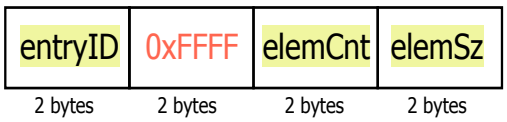

Figure 6. The structure of the Flash Copy of a RAM table entry

Where:

- **entryID** is the ID of the table entry
- **0xFFFF** has no meaning and is used for alignment purpose only

- **elemCnt** is the elements count of that entry
- **elemSz** is the size of a single element

The table marker has the value 0x3E42543C (“<TB>” if read as ASCII codes) and marks the beginning and the end of the NV table copy.

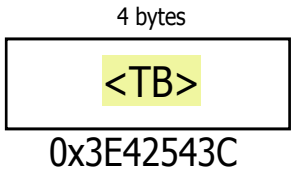

Figure 7. Table marker

This copy in FLASH of the RAM table is used to determine if the RAM table has been changed (for example as a result of an OTA upgrade). This check is performed when the storage system is initialized. If the RAM table has been changed, the new RAM table is copied in the second FLASH virtual page, then all the datasets that are still valid (still exists in the new RAM table) are copied. The system is able to manage the cases when a dataset elements count has been changed. For a given valid RAM table entry, if the elements count is less than the elements count stored in FLASH virtual page, only the elements from FLASH that are still exists in RAM are copied to the other FLASH page. In the opposite situation, when the elements count from the RAM table entry is greater than the elements count stored on FLASH page, the missing elements are copied from the RAM table entry.

Next after the end of the RAM table copy, the Meta Information Tags (MITs) follows.

Every MIT is used to store information related to one record.

A MIT has the following structure:

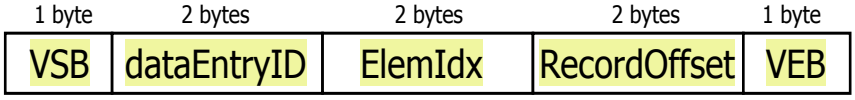

Figure 8. Meta Information Tag

Where:

- **VSB** is the validation start byte
- **dataEntryID** is the ID of the NV table entry
- **ElemIdx** is the element index
- **RecordOffset** is the offset of the record related to the start address of the virtual page
- **VEB** is the validation end byte

A valid MIT has VSB and VEB equals. If the MIT refers a single element record type, then **VSB=VEB=0xAA**. If the MIT refers a full table entry record type (all elements from a table entry), then **VSB=VEB=0x55**.

As the records are written to FLASH page, the page available space decreases. As a result, at one moment of time the page becomes full, i.e. a new record has not enough free space to be copied into that page. In the example presented below, the virtual page 1 is considered to be full if a new save request is pending and the page free space is not enough to copy the new record and the additional MIT. In such a case, the latest saved datasets (table entries) will be copied to virtual page 2.

In the following example, there are 5 datasets (one color for each dataset); to be a suggestive example, there are both ‘full’ and ‘single’ record types.

- **r1** is a ‘full’ record type (contains all the NV table entry elements) whereas **r3**, **r4** and **r11** are ‘single’ record types. Similar,
- **r2** - full record type; **r15** - single record type
- **r5**, **r13** - full record type; **r10**, **r12** - single record type
- **r8** - full record type;
- **r7**, **r9**, **r14**, **r16** - full record type

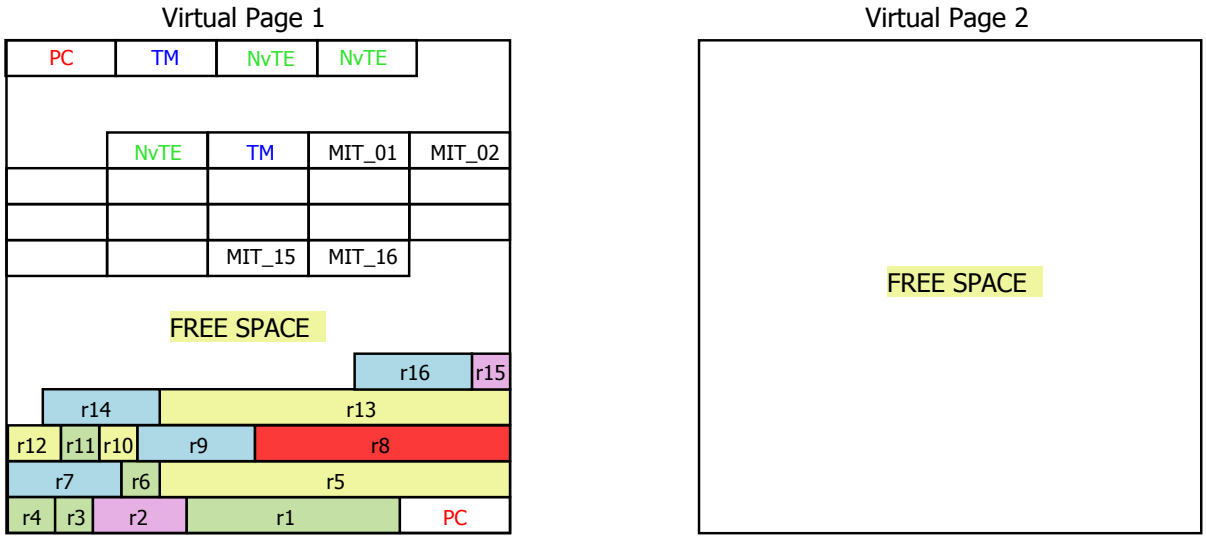

Figure 9. Virtual Page 1 free space is not enough to save a new dataset

As can be easily observed **r3**, **r4**, **r6** and **r11** are ‘single record’ types while **r1** is a ‘full record’ type of the same dataset. When copied to virtual page 2, a defragmentation process is taking place. As a result, the record copied to virtual page 2 has as much elements as **r1** but individual elements are taken from **r3, r4**, **r6** and **r11**.

After the copy process completes, the virtual page 2 has 5 ‘full record’ types, one for each dataset.

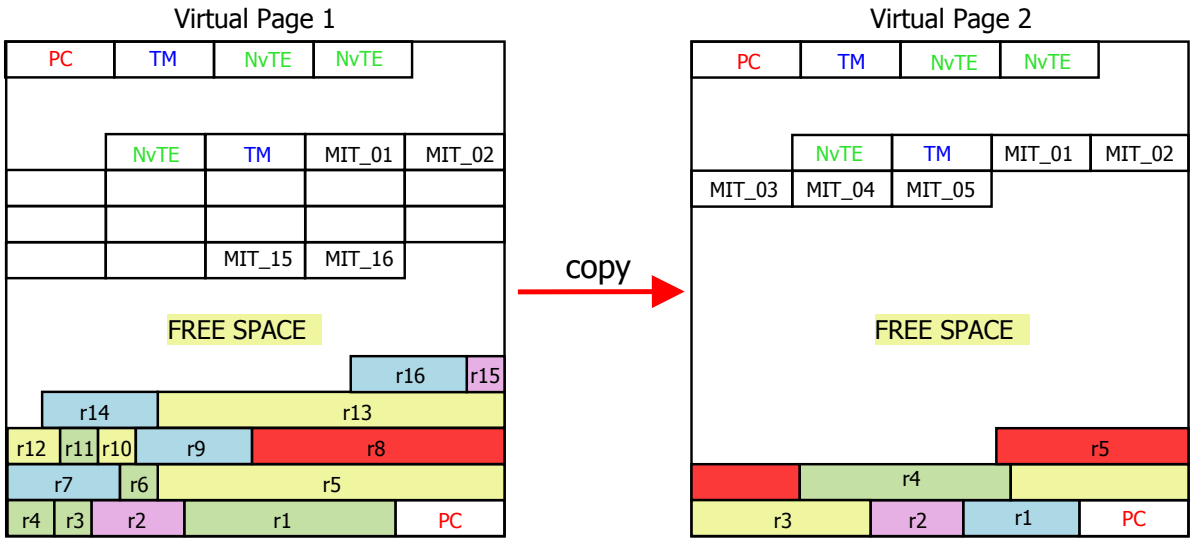

Figure 10. Page copy

Finally, the virtual page 2 is validated by writing the PC value and a request to erase virtual page 1 is performed. The page will be actually erased on idle task, sector by sector (only one sector is erased at a time, when idle task is executed).

3.5.3 Flex NVM

Several Kinetis MCU based platforms implement the Flex-Memory technology, with up to 512 KB of FlexNVM and up to 16 KB of FlexRAM. FlexNVM can be partitioned to support a simulated EEPROM sub-system. All the read/write operations are done in FlexRAM area and everytime a write operation occurs in this area, a new EEPROM backup record is produced and stored in FlexNVM. The EEPROM endurance capability may exceed 10 million cycles.

The Non-Volatile Memory platform component supports this feature and implements the associated routines. From the user point-of-view, the API is exactly the same. All the user has to do is to enable the FlexNVM support, by the mean of the **gNvUseFlexNVM_d** compiler switch.

The records and the associated Meta Information Tags are written to FlexRAM memory window. Each time a write/modify action is performed within FlexRAM, a new backup copy is produced and stored in FlexNVM memory. The wear-leveling algorithm is implemented and controlled by the hardware and it is not under user control. The Meta Information Tags stores information about the ID of the NV table entry and the offset to record itself, related to FlexRAM start address. After all datasets (RAM table entries) are written to FlexRAM, if an element of a given dataset (or the entire dataset) is changed and the upper layer decide that the change must be saved, the corresponding record is overwritten, partially or totally. In the below example, there are 8 NV table entries (datasets) saved on FlexRAM. ‘**meta08’** refers to ‘**rec08’** record and through information stored by the ‘**meta08’**, the entire record ‘**rec08’** can be read/updated, or just a single element (‘**e5’** in the below example).

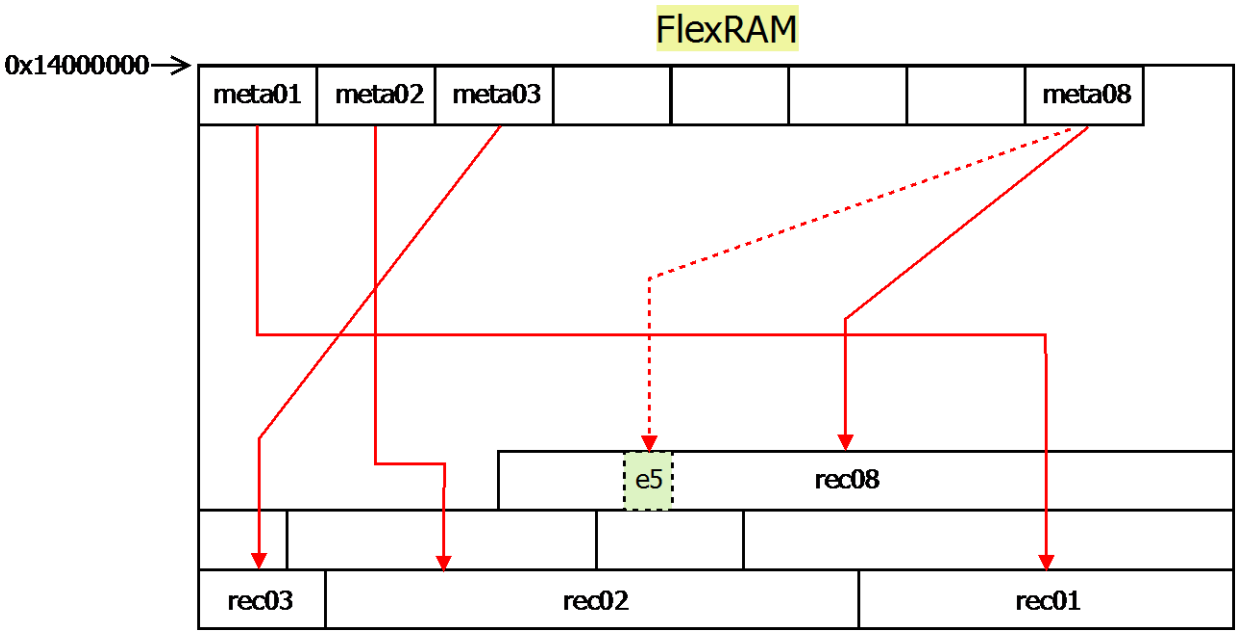

Figure 11. FlexRAM memory used to store NV table entries

3.5.4 Constant Macro Definitions

Name:

```
#define gNvStorageIncluded_d    FALSE
```

Description:

If set to TRUE it enables the whole functionality of the Non-volatile storage system. Default is set to FALSE (no code or data is generated for this module).

Name:

```
#define gNvUseFlexNVM_d    FALSE
```

Description:

If set to TRUE it enables the FlexNVM functionality of the Non-volatile storage system. Default is set to FALSE. If FlexNVM is used, the standard NV storage system is disabled.

Name:

```
#define gNvFragmentation_Enabled_d    FALSE
```

Description:

Macro used to enable/disable the fragmented saves/restores, i.e. a particular element from a table entry can be saved or restored. Default set to FALSE.

Name:

```
#define gNvUseExtendedFeatureSet_d    FALSE
```

Description:

Macro used to enable/disable the extended feature set of the module:

- Remove existing NV table entries
- Register new NV table entries
- Dynamic NV RAM tables
- Default set to FALSE.

Name:

```
#define gNvTableEntriesCountMax_c       32
```

Description:

This constant defines the maximum count of the table entries (datasets) that the application is going to use.

Name:

```
#define gNvRecordsCopiedBufferSize_c    64
```

Description:

This constant defines the size of the buffer used by the page copy function, when the copy operation performs defragmentation.

Name:

```
#define gNvCacheBufferSize_c            64
```

Description:

This constant defines the size of the cache buffer used by the page copy function, when the copy operation does not perform defragmentation.

Name:

```
#define gNvMinimumTicksBetweenSaves_c   4
```

Description:

This constant defines the minimum timer ticks between dataset saves (in seconds).

Name:

```
#define gNvCountsBetweenSaves_c         256
```

Description:

This constant defines the number of calls to ‘NvSaveOnCount’ between dataset saves.

Name:

```
#define gNvInvalidDataEntry_c           0xFFFFU
```

Description:

Macro used to mark a table entry as invalid in the NV table.

Name:

```
#define gNvFormatRetryCount_c           3
```

Description:

Macro used to define the maximum retries count value for the format operation.

Name:

```
#define gNvPendigSavesQueueSize_c       32
```

Description:

Macro used to define the size of the pending saves queue.

Name:

```
#define gNvEndOfTableId_c               0xFFFEU
```

Description:

Macro used to define the ID of the end-of-table entry.

Name:

```
#define gNvTableMarker_c                0x3E42543CUL
```

Description:

Macro used to define the table marker value. The table marker is used to indicate the start and the end of the FLASH copy of the NV table.

3.5.5 User defined data type definitions

Name:

```
typedef uint16_t NvSaveInterval_t;
```

Description:

Data type definition used by dataset saves on interval function.

Name:

```
typedef uint16_t NvSaveCounter_t;
```

Description:

Data type definition used by dataset saves on count function.

Name:

```
typedef uint16_t NvTableEntryId_t;
```

Description:

Data type definition for table entry ID.

Name:

```
typedef struct NVM_DatasetInfo_tag
{
    bool_t saveNextInterval;
    NvSaveInterval_t ticksToNextSave;
    NvSaveCounter_t countsToNextSave;
}NVM_DatasetInfo_t;
```

Description:

Data type definition for a dataset (NV table entry) information.

Name:

```
typedef struct NVM_DataEntry_tag
{
    void* pData;
    uint16_t ElementsCount;
    uint16_t ElementSize;
    uint16_t DataEntryID;
    uint16_t DataEntryType;
} NVM_DataEntry_t;
```

Description:

Data type definition for a NV table entry.

Name:

```
typedef struct NVM_Statistics_tag
{
    uint32_t FirstPageEraseCyclesCount;
    uint32_t SecondPageEraseCyclesCount;
} NVM_Statistics_t;
```

Description:

Data type definition used to store virtual pages statistic information.

Name:

```
typedef enum NVM_Status_tag
{
    gNVM_OK_c,
    gNVM_Error_c,
    gNVM_InvalidPageID_c,
    gNVM_PageIsNotBlank_c,
    gNVM_SectorEraseFail_c,
    gNVM_NullPointer_c,
    gNVM_PointerOutOfRange_c,
    gNVM_AddressOutOfRange_c,
    gNVM_InvalidSectorsCount_c,
    gNVM_InvalidTableEntry_c,
    gNVM_PageIsEmpty_c,
    gNVM_MetaNotFound_c,
    gNVM_RecordWriteError_c,
    gNVM_MetaInfoWriteError_c,
    gNVM_ModuleNotInitialized_c,
    gNVM_CriticalSectionActive_c,
    gNVM_ModuleAlreadyInitialized_c,
    gNVM_PageCopyPending_c,
    gNVM_RestoreFailure_c,
    gNVM_FormatFailure_c,
    gNVM_RegisterFailure_c,
    gNVM_AlreadyRegistered,
    gNVM_EraseFailure_c,
    gNVM_SaveRequestRejected_c,
    gNVM_InvalidTimerID_c,
    gNVM_MissingEndOfTableMarker_c,
    gNVM_NvTableExceedFlexRAMSize_c,
    gNVM_NvTableWrongElementSize_c,
    gNVM_NvWrongFlashDataIFRMap_c
} NVM_Status_t;
```

Description:

Enumerated data type definition for NV storage module error codes.

3.5.6 Flash Management API Primitives

3.5.6.1 NvModuleInit

Prototype:

```
NVM_Status_t NvModuleInit
(
  void
);
```

Description:

This function is used to initialize the Flash Management module. The function indirectly initializes the Flash HAL driver, gets the active page ID, checks if NV table has been changed and performs a page copy if the NV table was updated. It also initializes internal state variables and counters.

Parameters:

None

Returns:

The status of the initialization:

```
gNVM_OK_c
gNVM_FormatFailure_c
gNVM_ModuleAlreadyInitialized_c
gNVM_InvalidSectorsCount_c
gNVM_NvWrongFlashDataIFRMap_c
gNVM_MissingEndOfTableMarker_c
```

3.5.6.2 NvSaveOnIdle

Prototype:

```
NVM_Status_t NvSaveOnIdle
(
  void* ptrData,
  bool_t saveAll
);
```

Description:

This function saves the element or the entire NV table entry (dataset) pointed by the ptrData argument as soon as the NVM_Task() is the highest priority ready-to-run task in the system.

Parameters:

Name	Type	Direction	Description
ptrData	void*	IN	A pointer to the element or the table entry to be saved
saveAll	bool_t	IN	A flag used to specify if the entire table entry shall be saved or just the element pointed by ptrData

Returns:

One of the following:

```
gNVM_OK_c
gNVM_ModuleNotInitialized_c
gNVM_NullPointer_c
gNVM_InvalidTableEntry_c
gNVM_SaveRequestRejected_c
```

3.5.6.3 NvSaveOnInterval

Prototype:

```
NVM_Status_t NvSaveOnInterval
(
  void* ptrData
);
```

Description:

This function saves the specified dataset no often than a given time interval. If it has been at least that long since the last save, this function will cause a save as soon as the NVM_Task() is the highest priority ready-to-run task in the system.

Parameters:

Name	Type	Direction	Description
ptrData	void*	IN	A pointer to the table entry to be saved

Returns:

```
gNVM_OK_c
gNVM_ModuleNotInitialized_c
gNVM_NullPointer_c
gNVM_InvalidTableEntry_c
gNVM_SaveRequestRejected_c
```

3.5.6.4 NvSaveOnCount

Prototype:

```
NVM_Status_t NvSaveOnCount
(
  void* ptrData
);
```

Description:

This function increments a counter that is associated with the dataset specified by the function argument and when that counter equals or exceeds a trigger value, the dataset will be saved as soon as the NVM_Task() is the highest priority ready-to-run task in the system.

Parameters:

Name	Type	Direction	Description
ptrData	void*	IN	A pointer to the table entry to be saved

Returns:

None.

3.5.6.5 NvSetMinimumTicksBetweenSaves

Prototype:

```
void NvSetMinimumTicksBetweenSaves
(
  NvSaveInterval_t newInterval
);
```

Description:

This function sets a new value of the timer interval that is used by the “save on interval” mechanism. The change takes effect after the next save.

Parameters:

Name	Type	Direction	Description
newInterval	NvSaveInterval_t	IN	The new value to be applied to “save on interval” functionality

Returns:

None.

3.5.6.6 NvSetCountsBetweenSaves

Prototype:

```
void NvSetCountsBetweenSaves
(
  NvSaveCounter_t newCounter
);
```

Description:

This function sets a new value of the counter trigger that is used by the “save on count” mechanism. The change takes effect after the next save.

Parameters:

Name	Type	Direction	Description
newCounter	NvSaveCounter_t	IN	The new value to be applied to “save on count” functionality

Returns:

None.

3.5.6.7 NvTimerTick

Prototype:

```
bool_t NvTimerTick
(
  bool_t countTick
);
```

Description:

The function processes NvSaveOnInterval() requests. If the call of this function should count a timer tick, call it with countTick set to TRUE. Otherwise, call it with countTick set to FALSE. Regardless of the value of countTick, NvTimerTick() returns TRUE if one or more of the data sets' tick counters have not yet counted down to zero, or FALSE if all data set tick counters have reached zero. If NvTimerTick() returns TRUE, the timer should be ON. If NvTimerTick() returns FALSE, the timer can be turned off.

Parameters:

Name	Type	Direction	Description
countTick	bool_t	IN	See API description

Returns:

See description.

3.5.6.8 NvRestoreDataSet

Prototype:

```
NVM_Status_t NvRestoreDataSet
(
  void* ptrData,
  bool_t restoreAll
);
```

Description:

This function restores the element or the entire NV table entry specified by the function argument ptrData. If a valid table entry copy is found in flash memory, it will be restored to RAM NV Table.

Parameters:

Name	Type	Direction	Description
ptrData	void*	IN	A pointer to a NV table entry/element to be restored with data from Flash
restoreAll	bool_t	IN	A flag used to indicate if the entire table entry shall be restore or just a single element (indicated by ptrData)

Returns:

```
gNVM_OK_c
gNVM_ModuleNotInitialized_c
gNVM_NullPointer_c
gNVM_PointerOutOfRange_c
gNVM_PageIsEmpty_c
gNVM_Error_c
```

3.5.6.9 NvSetCriticalSection

Prototype:

```
void NvSetCriticalSection
(
  void
);
```

Description:

The function increments an internal counter variable each time it is called. All the save/erase/copy functions are checking this counter before execute their code. If the counter has a non-zero value, the function returns with no further operations.

Parameters:

None

Returns:

None.

3.5.6.10 NvClearCriticalSection

Prototype:

```
void NvClearCriticalSection
(
  void
);
```

Description:

The function decrements an internal counter variable each time it is called. All the save/erase/copy functions are checking this counter before execute their code. If the counter has a non-zero value, the function returns with no further operations.

Parameters:

None

Returns:

None.

3.5.6.11 NvIdle

Prototype:

```
void NvIdle
(
  void
);
```

Description:

This function processes the NvSaveOnIdle() and NvSaveOnCount() requests. Called by the NVM task.

Parameters:

None

Returns:

None.

3.5.6.12 NvIsDataSetDirty

Prototype:

```
bool_t NvIsDataSetDirty
(
  void* ptrData
);
```

Description:

This function checks if the table entry specified by the function argument is dirty.

Parameters:

Name	Type	Direction	Description
ptrData	void*	IN	A pointer to NV table entry to be checked

Returns:

TRUE if the specified table entry is dirty (i.e. different from the last valid copy saved in flash) / FALSE otherwise.

3.5.6.13 NvGetPageStatistics

Prototype:

```
void NvGetPagesStatistics
(
  NVM_Statistics_t* ptrStat
);
```

Description:

Retrieves the virtual pages statistics, i.e. how many times each virtual page has been erased.

Parameters:

Name	Type	Direction	Description
ptrStat	NVM_Statistics_t*	OUT	A pointer to a memory location where the statistics will be stored

Returns:

None.

3.5.6.14 NvFormat

Prototype:

```
NVM_Status_t NvFormat
(
  void
);
```

Description:

This function performs a full format of both virtual pages. The page counter value is preserved during formatting.

Parameters:

Name	Type	Direction	Description
-	-	-	-

Returns:

One of the following:

```
gNVM_OK_c
gNVM_ModuleNotInitialized_c
gNVM_CriticalSectionActive_c
gNVM_FormatFailure_c
```

3.5.6.15 NvRegisterTableEntry

Prototype:

```
NVM_Status_t NvRegisterTableEntry
(
  void* ptrData,
  NvTableEntryId_t uniqueId,
  uint16_t elemCount,
  uint16_t elemSize,
  bool_t overwrite
);
```

Description:

This function allows the user to register a new table entry or to update an existing one. To register a new table entry, the NV table must contain at least one invalid entry (e.g. a previously erased table entry).

Parameters:

Name	Type	Direction	Description
ptrData	void*	IN	A pointer to the table entry to be registered/updated
uniqueId	NvTableEntryId_t	IN	The ID of the table entry to be registered/updated
elemCount	uint16_t	IN	The elements count if the table entry to be registered/updated
elemSize	uint16_t	IN	The size of one element of the table entry
overwrite	bool_t	IN	If set to TRUE and the table entry ID already exists, the table entry will be updated with data provided by the function arguments

Returns:

One of the following:

```
gNVM_OK_c
gNVM_ModuleNotInitialized_c
gNVM_AlreadyRegistered
gNVM_RegisterFailure_c
```

3.5.6.16 NvEraseEntryFromStorage

Prototype:

```
NVM_Status_t NvEraseEntryFromStorage
(
  void* ptrData
);
```

Description:

This function removes the table entry specified by the function argument, ptrData.

Parameters:

Name	Type	Direction	Description
ptrData	void*	IN	A pointer to the table entry to be removed

Returns:

One of the following:

```
gNVM_OK_c
gNVM_ModuleNotInitialized_c
gNVM_CriticalSectionActive_c
gNVM_PointerOutOfRange_c
gNVM_NullPointer_c
gNVM_InvalidTableEntry_c
```

3.5.6.17 NvSyncSave

Prototype:

```
NVM_Status_t NvSyncSave
(
  void* ptrData,
  bool_t saveAll,
  bool_t ignoreCriticalSectionFlag
);
```

Description:

The function saves the pointed element or the entire table entry to the storage system. The save operation is not performed on the idle task but within this function call.

Parameters:

Name	Type	Direction	Description
ptrData	void*	IN	A pointer to the table entry to be saved
saveAll	bool_t	IN	Specifies if the entire table entry shall be saved or just the pointed element
ignoreCriticalSectionFlag	bool_t	IN	If set to TRUE, the function will ignore the critical section flag and will perform the save operation regardless of the value of the critical section flag.

Returns:

One of the following:

```
gNVM_OK_c
gNVM_ModuleNotInitialized_c
gNVM_CriticalSectionActive_c
gNVM_PointerOutOfRange_c
gNVM_NullPointer_c
```

3.5.6.18 NvAtomicSave

Prototype:

```
NVM_Status_t NvAtomicSave
(
  bool_t ignoreCriticalSectionFlag
);
```

Description:

The function performs an atomic save of the entire NV table to the storage system. All the required save operations are performed in place.

Parameters:

Name	Type	Direction	Description
ignoreCriticalSectionFlag	bool_t	IN	If set to TRUE, the function will ignore the critical section flag and will perform the save operation regardless of the value of the critical section flag.

Returns:

One of the following:

```
gNVM_OK_c
gNVM_ModuleNotInitialized_c
gNVM_CriticalSectionActive_c
gNVM_PointerOutOfRange_c
gNVM_NullPointer_c
```

3.6 Random Number Generator

3.6.1 Overview

The RNG module is part of the framework and is used for random number generation. It uses hardware RNG peripherals, 802.15.4 PHY RNG module and a software pseudo-random number generation algorithm. If no HW acceleration is present, the RNG module will use a SW algorithm! The initial seed for this algorithm represents the device unique ID (SIM_UID registers) by default. One can use the 802.15.4 PHY RNG for the initial seed, by setting the *gRNG_UsePhyRngForInitialSeed_d* define to 1.

3.6.2 Constant Macro Definitions

Name:

```
#define gRNG_HWSupport_d           0
#define gRNG_RNGAHWSupport_d       1
#define gRNG_RNGBHWSupport_d       2
#define gRNG_TRNGHWSupport_d       3
```

Description:

All possible HW support options for the RNG.

Name:

```
#ifndef gRNG_HWSupport_d
#define gRNG_HWSupport_d       gRNG_NoHWSupport_d
#endif
```

Description:

This macro defines the default HW support of the RNG module.

Name:

```
#define gRngSuccess_d           (0x00)
#define gRngInternalError_d     (0x01)
#define gRngNullPointer_d       (0x80)
```

Description:

Define the status codes for RNG.

Name:

```
#define gRngMaxRequests_d   (100000)
```

Description:

This macro defines the maximum number of requests permitted until a reseed is needed.

3.6.3 API Primitives

3.6.3.1 RNG_Init ()

Prototype:

```
uint8_t RNG_Init(void);
```

Description:
Initializes the hardware RNG module.

Parameters:

None.

Returns:

Status of the RNG module.

3.6.3.2 RNG_GetRandomNo ()

Prototype:

```
void RNG_GetRandomNo(uint32_t* pRandomNo)
```

Description:
Reads a random number from RNG module or from 802.15.4 PHY.

Parameters:

Name	Type	Direction	Description
pRandomNo	uint32_t *	[OUT]	Pointer to the location where the RNG will be stored.

Returns:

None.

3.6.3.3 RNG_SetPseudoRandomNoSeed ()

Prototype:

```
void RNG_SetPseudoRandomNoSeed(uint8_t* pSeed)
```

Description:
Initializes the seed for the PRNG algorithm.

Parameters:

Name	Type	Direction	Description
pSeed	uint8_t *	[IN]	Pointer to a buffer containing 20 bytes (160 bits).

Returns:

None.

3.6.3.4 RNG_GetPseudoRandomNo ()

Prototype:

```
int16_t RNG_GetPseudoRandomNo(uint8_t* pOut, uint8_t outBytes, uint8_t* pXSEED)
```

Description:
Pseudo Random Number Generator (PRNG) implementation according to NIST FIPS Publication 186-2, APPENDIX 3.

Parameters:

Name	Type	Direction	Description
pOut	uint8_t *	[OUT]	Pointer to the output buffer.
outBytes	uint8_t	[IN]	The number of bytes to be copied (1-20)
pXSEED	uint8_t *	[IN]	Optional user SEED. Should be NULL if not used.

Returns:

None.

3.7 System Panic

3.7.1 Overview

The framework provides a Panic function that halts system execution. When connected using a debugger, the execution stops at a ‘DEBUG’ instruction, which makes the Debugger stop and display the current program counter.

In the future the Panic function will also save relevant system data in Flash. Then an external tool will be able to retrieve the information from Flash so that the Panic cause can be investigated.

3.7.2 Constant Macro Definitions

Name:

```
#define ID_PANIC(grp,value) ((panicId_t)(((uint16_t)grp << 16)+((uint16_t)value)))
```

Description:

This macro creates the panic id, by concatenating an operation group with the operation value.

3.7.3 User defined data type definitions

Name:

```
typedef uint32_t panicId_t;
```

Description:

Panic identification data type definition

Name:

```
typedef struct
{
    panicId_t id;
    uint32_t location;
    uint32_t extra1;
    uint32_t extra2;
    uint32_t cpsr_contents;   /* may not be used initially */
    uint8_t stack_dump[4];    /* initially just contain the contents of the LR */
} panicData_t;
```

Description:

Panic data type definition

3.7.4 System Panic API Primitives

3.7.4.1 panic()

Prototype:

```
void panic
(
    panicId_t id,
    uint32_t location,
    uint32_t extra1,
    uint32_t extra2
);
```

Description:
Halts program execution, by disabling the interrupts and entering an infinite loop. If a debugger is connected, the execution stops at a 'DEBUG' instruction, which makes the Debugger stop and display the current program counter.

Parameters:

Name	Type	Direction	Description
id	panicId_t	[IN]	a group and a value packed as 32 bit
location	uint32_t	[IN]	usually the address of the function calling the panic
extra1	uint32_t	[IN]	Provide details about the cause of the panic
extra2	uint32_t	[IN]	Provide details about the cause of the panic

Returns:

None

3.8 System Reset

3.8.1 Overview

The framework provides a reset function that is used to software reset the MCU.

3.8.2 API Primitives

3.8.2.1 ResetMCU()

Prototype:

```
void ResetMCU
(
    void
);
```

Description:
Resets the MCU.

Parameters:

None

Returns:

None

3.9 Serial Manager

3.9.1 Overview

The framework allows the usage of multiple serial interfaces (UART, USB, SPI, IIC) using the same API.

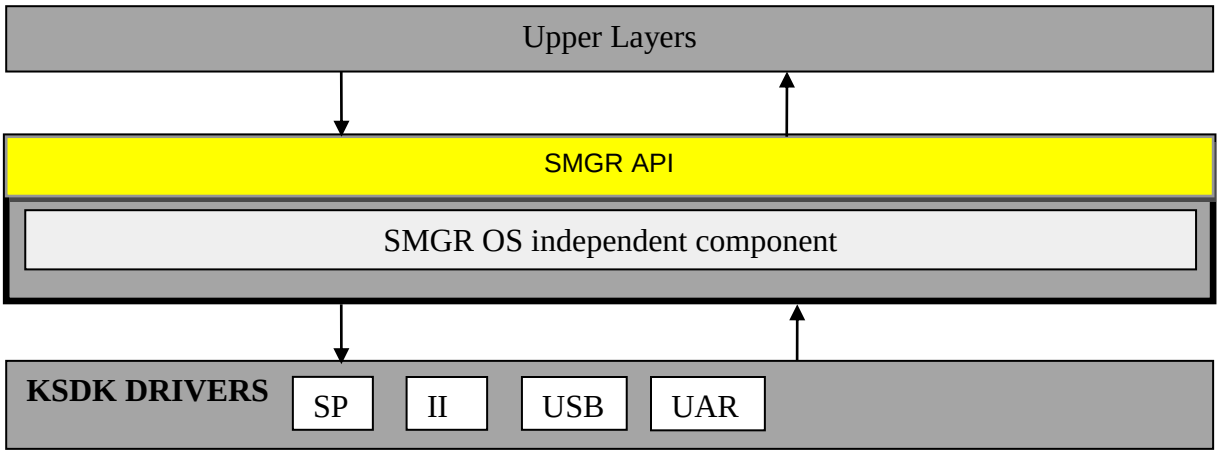

Figure 12. Serial Manager overview

Multiple interfaces can be used at the same time and can be defined on multiple peripherals of the same type.

When using asynchronous TX, a pointer to the data to be sent is stored in an internally managed buffer. This means that the user can call the API and then carry on. The TX callback will be executed when the TX operation finishes.

When using synchronous TX the user code will block until the TX operation is completed. Please not that a synchronous TX is also buffered and will complete, and unblock the user code, only after all the previous operations finish.

The user implemented TX callback can also use the serial manager API but some restrictions apply. Because callbacks are executed in the serial manager task context, no blocking calls shall be made. This includes synchronous TX. Please note that an asynchronous TX operation blocks if the internal buffer is full. If called from a callback, it will not block but instead an error message is returned which will be handled by the user.

3.9.2 Constant Macro Definitions

Name:

```
#define gSerialManagerMaxInterfaces_c 1
```

Description:

This define specifies the maximum number of interfaces to be used.

Name:

```
#define gSerialMgrUseUart_c 1
#define gSerialMgrUseUSB_c 0
#define gSerialMgrUseIIC_c 0
#define gSerialMgrUseSPI_c 0
```

Description:

Defines which serial interface can be used by SerialManager.

Name:

```
#define gSerialMgr_ParamValidation_d 1
```

Description:

Enables/Disabled input parameter checking.

Name:

```
#define gSerialMgr_BlockSenderOnQueueFull_c 1
```

Description:

Enables/Disabled blocking the calling task when an asynchronous TX operation is triggered with the queue full.

Name:

```
#define gSerialMgrIICAddress_c 0x76
```

Description:

Defines the address to be used for I2C.

Name:

```
#define gSerialMgrRxBufSize_c 32
```

Description:

Defines the RX buffer size.

Name:

```
#define gSerialMgrTxQueueSize_c 5
```

Description:

Defines the TX queue size.

Name:

```
#define gSerialTaskStackSize_c 1024
```

Description:

Defines the serial manager task stack size.

Name:

```
#define gSerialTaskPriority_c 3
```

Description:

Defines the serial manager task priority. Usually this task is a low priority task.

Name:

```
#define gPrtHexNoFormat_c  (0x00)
#define gPrtHexBigEndian_c (1<<0)
#define gPrtHexNewLine_c   (1<<1)
#define gPrtHexCommas_c    (1<<2)
#define gPrtHexSpaces_c    (1<<3)
```

Description:

If the user has to print a hex number he can choose between BigEndian=1/LittleEndian=0, newline, commas or spaces (between bytes).

3.9.3 Data type definitions

Name:

```
typedef enum{
    gSerialMgrNone_c,
    gSerialMgrUart_c,
    gSerialMgrUSB_c,
    gSerialMgrIICMaster_c,
    gSerialMgrIICSlave_c,
    gSerialMgrSPIMaster_c,
    gSerialMgrSPISlave_c
}serialInterfaceType_t;
```

Description:

Defines the types of serial interfaces.

Name:

```
typedef enum {
    gNoBlock_d     = 0,
    gAllowToBlock_d =1,
}serialBlock_t;
```

Description:

Defines if the TX is blocking or not.

Name:

```
typedef void (*pSerialCallBack_t)(void*);
```

Description:

Defines if the TX is blocking or not.

Name:

```
typedef enum{
    gUARTBaudRate1200_c   =   1200UL,
    gUARTBaudRate2400_c   =   2400UL,
    gUARTBaudRate4800_c   =   4800UL,
    gUARTBaudRate9600_c   =   9600UL,
    gUARTBaudRate19200_c  =  19200UL,
    gUARTBaudRate38400_c  =  38400UL,
    gUARTBaudRate57600_c  =  57600UL,
    gUARTBaudRate115200_c = 115200UL,
    gUARTBaudRate230400_c = 230400UL,
}serialUartBaudRate_t;
```

Description:

Defines the supported baud rates for UART.

Name:

```
typedef enum{
    gSPI_BaudRate_100000_c  = 100000,
    gSPI_BaudRate_200000_c  = 200000,
    gSPI_BaudRate_400000_c  = 400000,
    gSPI_BaudRate_800000_c  = 800000,
    gSPI_BaudRate_1000000_c = 1000000,
    gSPI_BaudRate_2000000_c = 2000000,
    gSPI_BaudRate_4000000_c = 4000000,
    gSPI_BaudRate_8000000_c = 8000000
}serialSpiBaudRate_t;
```

Description:

Defines the supported baud rates for SPI.

Name:

```
typedef enum{
    gIIC_BaudRate_50000_c  = 50000,
    gIIC_BaudRate_100000_c = 100000,
    gIIC_BaudRate_200000_c = 200000,
    gIIC_BaudRate_400000_c = 400000,
}serialIicBaudRate_t;
```

Description:

Defines the supported baud rates for IIC.

Name:

```
typedef enum{
   gSerial_Success_c,
   gSerial_InvalidParameter_c,
   gSerial_InvalidInterface_c,
   gSerial_MaxInterfacesReached_c,
   gSerial_InterfaceNotReady_c,
   gSerial_InterfaceInUse_c,
   gSerial_InternalError_c,
   gSerial_SemCreateError_c,
   gSerial_OutOfMemory_c,
   gSerial_OsError_c,
}serialStatus_t;
```

Description:

Serial manager status codes.

3.9.4 API Primitives

3.9.4.1 SerialManager_Init ()

Prototype:

```
void SerialManager_Init( void );
```

Description:
Creates the Serial Manager’s task and initializes internal data structures.

Parameters:

None.

Returns:

None.

3.9.4.2 Serial_InitInterface ()

Prototype:

```
serialStatus_t Serial_InitInterface (uint8_t *pInterfaceId,
                                     serialInterfaceType_t interfaceType,
                                     uint8_t channel);
```

Description:
Initialize a communication interface.

Parameters:

Name	Type	Direction	Description
InterfaceId	uint8_t *	[IN]	Interface ID.
interfaceType	serialInterfaceType_t	[IN]	The type of interface.
channel	uint8_t	[IN]	Channel number (required if MCU has more than one peripheral of the same type).

Returns:

- gSerial_InterfaceInUse_c if the interface is already opened
- gSerial_InvalidInterface_c if the interface is invalid
- gSerial_SemCreateError_c if semaphore creation fails
- gSerial_MaxInterfacesReached_c if the maximum number of interfaces has been reached
- gSerial_InternalError_c if an internal error occurred
- gSerial_Success_c if the operation was successful .

3.9.4.3 Serial_SetBaudRate ()

Prototype:

```
serialStatus_t Serial_SetBaudRate (uint8_t InterfaceId, uint32_t baudRate);
```

Description:
Sets the communication speed for an interface.

Parameters:

Name	Type	Direction	Description
InterfaceId	uint8_t	[IN]	Interface ID.
baudRate	uint32_t	[IN]	Communication speed.

Returns:

- gSerial_InvalidParameter_c if a parameter is invalid
- gSerial_InvalidInterface_c if the interface is invalid
- gSerial_InternalError_c if an internal error occurred
- gSerial_Success_c if the operation was successful .

3.9.4.4 Serial_RxBufferByteCount ()

Prototype:

```
serialStatus_t Serial_RxBufferByteCount (uint8_t InterfaceId, uint16_t *bytesCount);
```

Description:
Gets the number of bytes in the RX buffer.

Parameters:

Name	Type	Direction	Description
InterfaceId	uint8_t	[IN]	Interface ID.
bytesCount	uint16_t *	[OUT]	Number of bytes in the RX queue.

Returns:

- gSerial_InvalidParameter_c if a parameter is invalid
- gSerial_Success_c if the operation was successful .

3.9.4.5 Serial_SetRxCallBack ()

Prototype:

```
serialStatus_t Serial_SetRxCallBack (uint8_t InterfaceId, pSerialCallBack_t cb, void
*pRxParam);
```

Description:
Sets a pointer to a function that will be called when data is received.

Parameters:

Name	Type	Direction	Description
InterfaceId	uint8_t	[IN]	Interface ID.
cb	pSerialCallBack_t	[IN]	Pointer to the callback function.
pRxParam	void *	[IN]	

Returns:

- gSerial_InvalidParameter_c if a parameter is invalid
- gSerial_Success_c if the operation was successful .

3.9.4.6 Serial_Read ()

Prototype:

```
serialStatus_t Serial_Read (uint8_t InterfaceId,
                            uint8_t *pData,
                            uint16_t bytesToRead,
                            uint16_t *bytesRead);
```

Description:
Returns a specified number of characters from the Rx buffer.

Parameters:

Name	Type	Direction	Description
InterfaceId	uint8_t	[IN]	Interface ID.
pData	uint8_t *	[OUT]	Pointer to a buffer to store the data.
bytesToRead	uint16_t	[IN]	Number of bytes to read.
bytesRead	uint16_t *	[OUT]	Pointer to a location to store the number of bytes read.

Returns:

- gSerial_InvalidParameter_c if a parameter is invalid
- gSerial_InvalidInterface_c if the interface is invalid
- gSerial_Success_c if the operation was successful .

3.9.4.7 Serial_GetByteFromRxBuffer ()

Prototype:

```
#define Serial_GetByteFromRxBuffer(InterfaceId, pDst, readBytes) Serial_Read(InterfaceId, pDst, 1, readBytes)
```

Description:
Retrieves one byte from the Rx buffer. Returns the number of bytes retrieved (1/0).

Parameters:

Name	Type	Direction	Description
InterfaceId	uint8_t	[IN]	Interface ID.
pDst	uint8_t *	[OUT]	Output buffer pointer.
readBytes	uint16_t *	[OUT]	Output value representing the bytes retrieved (1 or 0).

Returns:

- gSerial_InvalidParameter_c if a parameter is invalid
- gSerial_Success_c if successful
- gSerial_InvalidInterface_c if the interface is not valid.

3.9.4.8 Serial_SyncWrite ()

Prototype:

```
serialStatus_t Serial_SyncWrite (uint8_t InterfaceId, uint8_t *pBuf, uint16_t bufLen);
```

Description:
Transmits a data buffer synchronously. The task will block until the TX is done.

Parameters:

Name	Type	Direction	Description
InterfaceId	uint8_t	[IN]	Interface ID.
pBuf	uint8_t *	[IN]	Pointer to a buffer containing the data to be sent.
bufLen	uint16_t	[IN]	Buffer length

Returns:

- gSerial_InvalidParameter_c if a parameter is invalid
- gSerial_OutOfMemory_c if there is no room left in the TX queue
- gSerial_Success_c if the operation was successful .

3.9.4.9 Serial_AsyncWrite ()

Prototype:

```
serialStatus_t Serial_AsyncWrite (uint8_t InterfaceId, uint8_t *pBuf, uint16_t bufLen,
                                  pSerialCallBack_t cb, void *pTxParam);
```

Description:
Transmit a data buffer asynchronously.

Parameters:

Name	Type	Direction	Description
InterfaceId	uint8_t	[IN]	Interface ID.
pBuf	uint8_t *	[IN]	Pointer to a buffer containing the data to be sent.
bufLen	uint16_t	[IN]	Buffer length
cb	pSerialCallBack_t	[IN]	Pointer to the callback function.
pTxParam	void *	[IN]	Parameter to be passed to the callback when it executes.

Returns:

- gSerial_InvalidParameter_c if a parameter is invalid
- gSerial_OutOfMemory_c if there is no room left in the TX queue
- gSerial_Success_c if the operation was successful .

3.9.4.10 Serial_Print ()

Prototype:

```
serialStatus_t Serial_Print (uint8_t InterfaceId, char * pString, serialBlock_t
allowToBlock);
```

Description:
Prints a string to the serial interface.

Parameters:

Name	Type	Direction	Description
InterfaceId	uint8_t	[IN]	Interface ID.
pString	char *	[IN]	Pointer to a buffer containing the string to be sent.
allowToBlock	serialBlock_t	[IN]	Specifies if the task will wait for the TX to finish or not.

Returns:

- gSerial_InvalidParameter_c if a parameter is invalid
- gSerial_OutOfMemory_c if there is no room left in the TX queue
- gSerial_Success_c if the operation was successful .

3.9.4.11 Serial_PrintHex ()

Prototype:

```
serialStatus_t Serial_PrintHex (uint8_t InterfaceId, uint8_t *hex, uint8_t len, uint8_t
flags);
```

Description:
Prints an number in hexadecimal format to the serial interface. The task will wait until the TX has finished.

Parameters:

Name	Type	Direction	Description
InterfaceId	uint8_t	[IN]	Interface ID.
hex	uint8_t *	[IN]	Pointer to the number to be printed.
len	uint8_t	[IN]	The number of bytesof the number
flags	uint8_t	[IN]	Flags specify display options: comma, space, new line

Returns:

- gSerial_InvalidParameter_c if a parameter is invalid
- gSerial_OutOfMemory_c if there is no room left in the TX queue
- gSerial_Success_c if the operation was successful .

3.9.4.12 Serial_PrintDec ()

Prototype:

```
serialStatus_t Serial_PrintDec (uint8_t InterfaceId, uint32_t nr);
```

Description:
Prints an unsigned integer to the serial interface.

Parameters:

Name	Type	Direction	Description
InterfaceId	uint8_t	[IN]	Interface ID.
nr	uint32_t	[IN]	Number to be printed.

Returns:

- gSerial_InvalidParameter_c if a parameter is invalid
- gSerial_OutOfMemory_c if there is no room left in the TX queue
- gSerial_Success_c if the operation was successful

3.9.4.13 Serial_EnableLowPowerWakeup ()

Prototype:

```
serialStatus_t Serial_EnableLowPowerWakeup( serialInterfaceType_t interfaceType);
```

Description:
Configures the enabled hardware modules of the given interface type as a wakeup source from STOP mode.

Parameters:

Name	Type	Direction	Description
interfaceType	serialInterfaceType_t	[IN]	Interface type of the modules to configure.

Returns:

- gSerial_Success_c if there is at least one module to configure
- gSerial_InvalidInterface_c otherwise.

3.9.4.14 Serial_DisableLowPowerWakeup ()

Prototype:

```
serialStatus_t Serial_DisableLowPowerWakeup( serialInterfaceType_t interfaceType);
```

Description:
Configures the enabled hardware modules of the given interface type as modules without wakeup capabilities.

Parameters:

Name	Type	Direction	Description
interfaceType	serialInterfaceType_t	[IN]	Interface type of the modules to configure.

Returns:

- gSerial_Success_c if there is at least one module to configure
- gSerial_InvalidInterface_c otherwise.

3.9.4.15 Serial_IsWakeUpSource ()

Prototype:

```
bool_t Serial_IsWakeUpSource( serialInterfaceType_t interfaceType);
```

Description:
Decides whether a enabled hardware module of the given interface type woke up the CPU from STOP mode.

Parameters:

Name	Type	Direction	Description
interfaceType	serialInterfaceType_t	[IN]	Interface type of the modules to be evaluated as wakeup source.

Returns:

TRUE if a module of the given interface type was the wakeup source FALSE otherwise.

3.9.4.16 HexToAscii ()

Prototype:

```
#define HexToAscii(hex)
```

Description:
Converts a 0x00-0x0F (4 bytes) number to ascii '0'-'F'.

Parameters:

Name	Type	Direction	Description
hex	uint8_t	[IN]	Number to be converted.

Returns:

Ascii code of the number.

3.10 FSCI

3.10.1 Overview

The Freescale Serial Communication Interface (FSCI) is at the same time a software module and a protocol that allows monitoring and extensive testing of the protocol layers interfaces. It also allows separation of the protocol stack between two protocol layers in a two processing entities setup: the host processor (typically running the upper layers of a protocol stack) and the Black Box application (typically containing the lower layers of the stack, serving as a modem). The Freescale Test Tool software is an example of a host processor which can interact with and FSCI Black Boxes at various layers. In this setup a user can run numerous commands to test the Black Box application services and interfaces.

The FSCI enables common service features for each device and allows monitoring of specific interfaces and API calls. Additionally, the FSCI injects or calls specific events and commands into the interfaces between layers.

An entity which needs to be interfaced to the FSCI module can use the API to register op codes to specific interfaces. After doing so, any packet coming from that interface with the same op code will trigger a callback execution. Two or more entities cannot register the same op code on the same interface, but they can do so for different interfaces. For example two MAC instances can register the same op codes, one over UARTA and the other over UARTB. This way we can use Test Tool to communicate to each MAC layer over two UART interfaces.

NOTE

The FSCI module executes in the context of the Serial Manager task.

3.10.2 FSCI packet structure

The FSCI module sends and receives messages as shown in the figure below. This structure is not specific to a serial interface and is designed to offer the best communication reliability. The Black Box device is expecting messages in little-endian format and responds with messages in little-endian format.

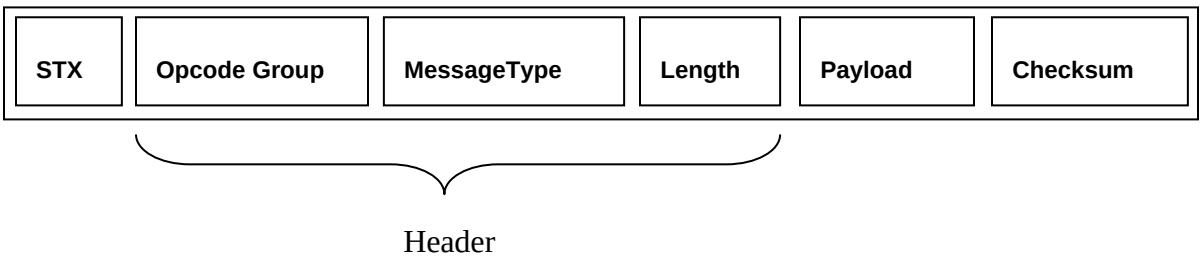

Figure 13. Packet Structure

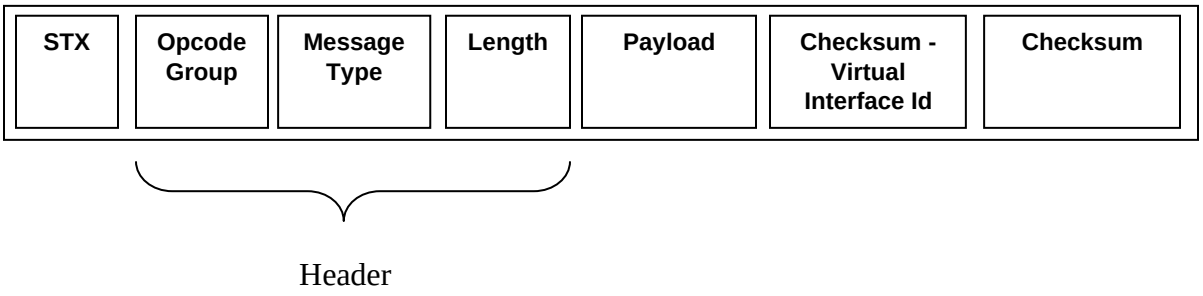

Figure 14. Packet Structure when virtual interfaces are used

Field Name	Length (bytes)	description
STX	1	Used for synchronization over the serial interface. The value is always 0x02.
Opcode Group	1	Distinguishes between different Service Access Primitives (e.g. MLME or MCPS)
Message Type	1	Specifies the exact message opcode that is contained in the packet
Length	1/2	The length of the packet payload, excluding the header and FCS. The length field content shall be provided in little endian format.
Payload	variable	Payload of the actual message.
Checksum	1	Checksum field used to check the data integrity of the packet.
Checksum2	0/1	The second CRC field appears only for virtual interfaces

Table 2 Packet Field Description

NOTE

When virtual interfaces are used, the first checksum is decremented with the Id of the interface, and the second checksum is used for error detection!

3.10.3 Constant Macro Definitions

Name:

```
#define gFsciIncluded_c           0 /* Enable/Disable FSCI module */
#define gFsciMaxOpGroups_c        8
```

```
#define gFsciMaxInterfaces_c      1
#define gFsciMaxVirtualInterfaces_c 2
#define gFsciMaxPayloadLen_c    245 /* bytes */
#define gFsciTimestampSize_c      0 /* bytes */
#define gFsciLenHas2Bytes_c       0 /* boolean */
#define gFsciUseEscapeSeq_c       0 /* boolean */
#define gFsciUseFmtLog_c          0 /* boolean */
#define gFsciUseFileDataLog_c     0 /* boolean */
#define gFsciLoggingInterface_c   1 /* [0..gFsciMaxInterfaces_c) */
```

Description:

Configures the FSCI module.

Name:

```
#define gFSCI_MlmeNwkOpcodeGroup_c       0x84  /* MLME_NWK_SapHandler          */
#define gFSCI_NwkMlmeOpcodeGroup_c       0x85  /* NWK_MLME_SapHandler          */
#define gFSCI_McpsNwkOpcodeGroup_c       0x86  /* MCPS_NWK_SapHandler          */
#define gFSCI_NwkMcpsOpcodeGroup_c       0x87  /* NWK_MCPS_SapHandler          */
#define gFSCI_AspAppOpcodeGroup_c        0x94  /* ASP_APP_SapHandler           */
#define gFSCI_AppAspOpcodeGroup_c        0x95  /* APP_ASP_SapHandler           */

#define gFSCI_LoggingOpcodeGroup_c       0xB0  /* FSCI data logging utillity    */
#define gFSCI_ReqOpcodeGroup_c           0xA3  /* FSCI utility Requests         */
#define gFSCI_CnfOpcodeGroup_c           0xA4  /* FSCI utility Confirmations/Indications
*/
#define gFSCI_ReservedOpGroup_c          0x52
```

Description:

The OpGroups reserved by MAC, App and FSCI.

3.10.4 Data type definitions

Name:

```
typedef enum{
    gFsciSuccess_c                = 0x00,
    gFsciSAPHook_c                = 0xEF,
    gFsciSAPDisabled_c            = 0xF0,
    gFsciSAPInfoNotFound_c        = 0xF1,
    gFsciUnknownPIB_c             = 0xF2,
    gFsciAppMsgTooBig_c           = 0xF3,
    gFsciOutOfMessages_c          = 0xF4,
    gFsciEndPointTableIsFull_c    = 0xF5,
    gFsciEndPointNotFound_c       = 0xF6,
    gFsciUnknownOpcodeGroup_c     = 0xF7,
    gFsciOpcodeGroupIsDisabled_c  = 0xF8,
    gFsciDebugPrintFailed_c       = 0xF9,
    gFsciReadOnly_c               = 0xFA,
    gFsciUnknownIBIdentifier_c    = 0xFB,
    gFsciRequestIsDisabled_c      = 0xFC,
    gFsciUnknownOpcode_c          = 0xFD,
    gFsciTooBig_c                 = 0xFE,
    gFsciError_c                  = 0xFF    /* General catchall error. */
} gFsciStatus_t;
```

Description:

FSCI status codes.

Name:

```
enum {
    mFsciMsgModeSelectReq_c                 = 0x00, /* Fsci-ModeSelect.Request */
    mFsciMsgGetModeReq_c                    = 0x02, /* Fsci-GetMode.Request */
    mFsciMsgResetCPUReq_c                   = 0x08, /* Fsci-CPU_Reset.Request */

    mFsciOtapSupportSetModeReq_c            = 0x28,
    mFsciOtapSupportStartImageReq_c         = 0x29,
    mFsciOtapSupportPushImageChunkReq_c     = 0x2A,
    mFsciOtapSupportCommitImageReq_c        = 0x2B,
    mFsciOtapSupportCancelImageReq_c        = 0x2C,
    mFsciOtaSupportSetFileVerPoliciesReq_c  = 0x2D,
    mFsciOtaSupportAbortOTAUpgradeReq_c     = 0x2E,
    mFsciOtapSupportImageChunkReq_c         = 0x2F,
    mFsciOtapSupportQueryImageReq_c         = 0xC2,
    mFsciOtapSupportQueryImageRsp_c         = 0xC3,
    mFsciOtapSupportImageNotifyReq_c        = 0xC4,

    mFsciLowLevelMemoryWriteBlock_c         = 0x30, /* Fsci-WriteRAMMemoryBlock.Request */
    mFsciLowLevelMemoryReadBlock_c          = 0x31, /* Fsci-ReadMemoryBlock.Request */
    mFsciLowLevelPing_c                     = 0x38, /* Fsci-Ping.Request */

    mFsciMsgAllowDeviceToSleepReq_c         = 0x40, /* Fsci-SelectWakeUpPIN.Request */
    mFsciMsgWakeUpIndication_c              = 0x41, /* Fsci-WakeUp.Indication */
    mFsciMsgReadExtendedAdrReq_c            = 0xD2, /* Fsci-ReadExtAddr.Request */
    mFsciMsgWriteExtendedAdrReq_c           = 0xDB, /* Fsci-WriteExtAddr.Request */

    mFsciMsgError_c                         = 0xFE, /* FSCI error message. */
    mFsciMsgDebugPrint_c                    = 0xFF, /* printf()-style debug message. */
};
```

Description:

Define the message types that FSCI recognizes and/or generates.

Name:

```
typedef void (*pfMsgHandler_t)(void* pData, void* param, uint32_t interfaceId);
```

Description:

Defines the message handler function type.

Name:

```
typedef gFsciStatus_t (*pfMonitor_t) (opGroup_t opGroup, void *pData, void* param, uint32_t
interfaceId);
```

Description:

Message Handler Function type definition.

Name:

```
typedef uint8_t clientPacketStatus_t;
```

Description:

FSCI response status code.

Name:

```
typedef uint8_t opGroup_t;
```

Description:

The operation group data type.

Name:

```
typedef uint8_t opCode_t;
```

Description:

The operation code data type.

Name:

```
#if gFsciLenHas2Bytes_c
typedef uint16_t fsciLen_t;
#else
typedef uint8_t fsciLen_t;
#endif
```

Description:

Payload length data type.

Name:

```
typedef struct gFsciOpGroup_tag
{
    pfMsgHandler_t pfOpGroupHandler;
    void*          param;
    opGroup_t      opGroup;
    uint8_t        mode;
    uint8_t        fsciInterfaceId;
} gFsciOpGroup_t;
```

Description:

Defines the Operation Group table entry.

Name:

```
typedef PACKED_STRUCT clientPacketHdr_tag
{
    uint8_t   startMarker;
    opGroup_t opGroup;
    opCode_t  opCode;
    fsciLen_t   len;      /* Actual length of payload[] */
} clientPacketHdr_t;
```

Description:

Format of packet header exchanged between the external client and FSCI.

Name:

```
typedef PACKED_STRUCT clientPacketStructured_tag
{
    clientPacketHdr_t header;
    uint8_t payload[gFsciMaxPayloadLen_c];
    uint8_t checksum;
} clientPacketStructured_t;
```

Description:

Format of packets exchanged between the external client and FSCI. The terminal checksum is actually stored at payload[len]. The checksum field insures that there is always space for it, even if the payload is full.

Name:

```
typedef PACKED_UNION clientPacket_tag
{
    /* The entire packet as unformatted data. */
    uint8_t raw[sizeof(clientPacketStructured_t)];
    /* The packet as header + payload. */
    clientPacketStructured_t structured;
    /* A minimal packet with only a status value as a payload. */
    PACKED_STRUCT
    {                             /* The packet as header + payload. */
        clientPacketHdr_t header;
        clientPacketStatus_t status;
    } headerAndStatus;
} clientPacket_t;
```

Description:

Format of packets exchanged between the external client and FSCI.

Name:

```
typedef enum{
    gFsciDisableMode_c,
    gFsciHookMode_c,
    gFsciMonitorMode_c,
    gFsciInvalidMode = 0xFF
```

```
} gFsciMode_t;
```

Description:

Defines the FSCI OpGroup operating mode.

Name:

```
typedef struct{
    uint32_t              baudrate;
    serialInterfaceType_t interfaceType;
    uint8_t               interfaceChannel;
    uint8_t               virtualInterface;
} gFsciSerialConfig_t;
```

Description:

FSCI Serial Interface initialization structure.

3.10.5 FSCI API Primitives

3.10.5.1 FSCI_Init ()

Prototype:

```
void FSCI_Init(void* argument);
```

Description:
Initializes the FSCI internal variables.

Parameters:

Name	Type	Direction	Description
interfaceType	serialInterfaceType_t	[IN]	Argument pointer to a initialization structure.

Returns:

None.

3.10.5.2 FSCI_RegisterOpGroup

Prototype:

```
gFsciStatus_t FSCI_RegisterOpGroup (opGroup_t opGroup, gFsciMode_t mode,
                                    pfMsgHandler_t pHandler,
                                    void* param,
                                    uint32_t fsciInterface);
```

Description:
Registers a message handler function for the specified Operation Group.

Parameters:

Name	Type	Direction	Description
OG	opGroup_t	[IN]	The Operation Group
mode	gFsciMode_t	[IN]	The operating mode
pHandler	pfMsgHandler_t	[IN]	Pointer to a function that will handle the received message
param	void*	[IN]	Pointer to a parameter that will be provided inside the in the OG Handler function
fsciInterface	uint32_t	[IN]	The interface ID on which the callback should be registered

Returns:

- gFsciSuccess_c if the operation was successful
- gFsciError_c if there is no more space in the table or the OG specified already exists.

3.10.5.3 FSCI_Monitor

Prototype:

```
gFsciStatus_t FSCI_Monitor (opGroup_t opGroup, void *pData, void* param, uint32_t
fsciInterface);
```

Description:

This function is used for monitoring SAPs.

Parameters:

Name	Type	Direction	Description
opGroup	opGroup_t	[IN]	The operation group
pData	uint8_t*	[IN]	Pointer to data location
param	uint32_t	[IN]	A parameter that will be passed to the OG Handler function (ex: a status message)
fsciInterface	uint32_t	[IN]	The interface on which the data should be printed

Returns:

Returns the status of the call process.

3.10.5.4 FSCI_LogToFile

Prototype:

```
void FSCI_LogToFile (char *fileName, uint8_t *pData, uint16_t dataSize, uint8_t mode);
```

Description:
Sends binary data to a specific file.

Parameters:

Name	Type	Direction	Description
filename	char	[IN]	The name of the file in which the data will be stored.
pData	uint8_t*	[IN]	Pointer to the data to be written.
dataSize	uint16_t	[IN]	The size of the data to be written.
mode	uint8_t	[IN]	The mode in which the file will be accessed.

Returns:

None.

3.10.5.5 FSCI_LogFormattedFile

Prototype:

```
void FSCI_LogFormatedText (const char *fmt, ...);
```

Description:
Sends a formated text string to the host.

Parameters:

Name	Type	Direction	Description
fmt	char *	[IN]	The string and format specifiers to output to the datalog.
...	any	[IN]	The variable number of parameters to output to the datalog.

Returns:

None.

3.10.5.6 FSCI_Print

Prototype:

```
void FSCI_Print(uint8_t readyToSend, void *pSrc, fsciLen_t len);
```

Description:
Sends a byte string over the serial interface.

Parameters:

Name	Type	Direction	Description
readyToSend	uint8_t	[IN]	Specify if the data should be transmitted asap.
pSrc	void*	[IN]	Pointer to the data location

len	index_t	[IN]	Size of the data

Returns:

None.

3.10.5.7 FSCI_ProcessRxPkt

Prototype:

```
gFsciStatus_t FSCI_ProcessRxPkt (clientPacket_t* pPacket, uint32_t fsciInterface);
```

Description:
Sends a message to the FSCI module.

Parameters:

Name	Type	Direction	Description
pPacket	clientPacket_t *	[IN]	A pointer to the message payload.
fsciInterface	uint32_t	[IN]	The interface on which the data was received.

Returns:

The status of the operation.

3.10.5.8 FSCI_CallRegisteredFunc

Prototype:

```
gFsciStatus_t FSCI_CallRegisteredFunc (opGroup_t opGroup, void *pData, uint32_t
fsciInterface);
```

Description:
This calls the handler for a specific OpGroup.

Parameters:

Name	Type	Direction	Description
opGroup	opGroup	[IN]	The OpGroup of the message.
pData	void *	[IN]	A pointer to the message payload.
fsciInterface	uint32_t	[IN]	The interface on which the data should be printed.

Returns:

Returns the status of the call process.

3.10.5.9 FSCI_transmitFormatedPacket

Prototype:

```
void FSCI_transmitFormatedPacket( void *pPacket, uint32_t fsciInterface );
```

Description:
Send packet over the serial interface, after computing Checksum.

Parameters:

Name	Type	Direction	Description
pPacket	void *	[IN]	Pointer to the packet to be sent over the serial interface.
fsciInterface	uint32_t	[IN]	The interface on which the packet should be sent.

Returns:

None.

3.10.5.10 FSCI_transmitPayload

Prototype:

```
void FSCI_transmitPayload(uint8_t OG, uint8_t OC, void * pMsg, uint16_t size, uint32_t
interfaceId);
```

Description:
Encode and send messages over the serial interface.

Parameters:

Name	Type	Direction	Description
OG	uint8_t	[IN]	Operation Group.
OC	uint8_t	[IN]	Operation Code.
pMsg	void *	[IN]	Pointer to payload.
size	uint16_t	[IN]	Length of the payload.
interfaceId	uint32_t	[IN]	The interface on which the packet should be sent.

Returns:

None.

3.10.5.11 FSCI_Error

Prototype:

```
void FSCI_Error( uint8_t errorCode, uint32_t fsciInterface );
```

Description:
Send packet over the serial interface with the specified error code.

This function does not use dynamic memory, and the packet is sent in blocking mode.

Parameters:

Name	Type	Direction	Description
errorCode	uint8_t	[IN]	The FSCI error code to be transmitted.
fsciInterface	uint32_t	[IN]	The interface on which the packet should be sent.

Returns:

None.

3.10.6 FSCI usage example

Initialization

```
/* Configure the number of interfaces and virtual interfaces used */
#define gFsciMaxInterfaces_c             4
#define gFsciMaxVirtualInterfaces_c      2

….
/* Define the interfaces used */
static const gFsciSerialConfig_t myFsciSerials[] = {
    /* Baudrate,            interface type,   channel No, virtual interface */
    {gUARTBaudRate115200_c, gSerialMgrUart_c,     1,           0},
    {gUARTBaudRate115200_c, gSerialMgrUart_c,     1,           1},
    {0                    , gSerialMgrIICSlave_c, 1,           0},
    {0                    , gSerialMgrUSB_c,      0,           0},
};

….
/* Call init function to open all interfaces */
FSCI_Init( (void*)mFsciSerials );
```

Registering Operation Groups

```
myOpGroup = 0x12; // Operation Group used
myParam = NULL; // pointer to a parameter to be passed to the handler function
(myHandlerFunc)
myInterface = 1; // index of entry from myFsciSerials
…
FSCI_RegisterOpGroup( myOpGroup, gFsciMonitorMode_c, myHandlerFunc, myParam, myInterface );
```

Implementing handler function

```
void fsciMcpsReqHandler(void *pData, void* param, uint32_t interfaceId)
{
```

```
    clientPacket_t     *pClientPacket = ((clientPacket_t*)pData);
    fsciLen_t myNewLen;

    switch( pClientPacket->structured.header.opCode ) {
    case 0x01:
    {
        /* Reuse packet received over the serial interface
           The OpCode remains the same.
           The length of the response must be <= that the length of the received packet */
        pClientPacket->structured.header.opGroup = myResponseOpGroup;
/* Process packet */
 …
        pClientPacket->structured.header. len = myNewLen;
        FSCI_transmitFormatedPacket(pClientPacket, interfaceId);
        return;
    }

    case 0x02:
    {
        /* Alocate a new message for the response.
           The received packet is Freed */
        clientPacket_t *pResponsePkt = MEM_BufferAlloc( sizeof(clientPacketHdr_t) +
                                                        myPayloadSize_d +
                                                        sizeof(uint8_t) // CRC
                                                      );

        if(pResponsePkt)
        {
            /* Process received data and fill the response packet */
            …
            pResponsePkt->structured.header. len = myPayloadSize_d;
            FSCI_transmitFormatedPacket(pClientPacket, interfaceId);
        }
        break;
    }

    default:
        MEM_BufferFree( pData );
        FSCI_Error( gFsciUnknownOpcode_c, interfaceId );
        return;
    }

    /* Free message received over the serial interface */
    MEM_BufferFree( pData );
}
```

3.11 Sec Lib

3.11.1 Overview

The framework provides support for cryptography in the security module. It supports both software and hardware encryption. The hardware encryption uses the MMCAU instruction set. Using the hardware support directly requires the input data to by 4 bytes aligned.

Both implementations are supplied in library format.

3.11.2 Constant Macro Definitions

Name:

```
#define gSecLib_NoHWSupport_d  0
#define gSecLib_MMCAUSupport_d 1
#define gSecLib_LTCSupport_d   2
```

Description:

Defines all possible HW support options for SecLib.

Name:

```
#ifndef gSecLib_HWSupport_d
#define gSecLib_HWSupport_d  gSecLib_NoHWSupport_d
#endif
```

Description:

Defines the default HW support option.

3.11.3 Data type definitions

Name:

```
typedef enum
{
    gSecSuccess_c,
    gSecAllocError_c,
    gSecError_c
} secResultType_t;
```

Description:

The status of the AES functions

Name:

```
typedef struct sha1Context_tag{
    uint32_t hash[SHA1_HASH_SIZE/sizeof(uint32_t)];
    uint8_t  buffer[SHA1_BLOCK_SIZE];
    uint32_t totalBytes;
    uint8_t  bytes;
```

```
}sha1Context_t;
```

Description:

The context used by the SHA1 functions

Name:

```
typedef struct sha256Context_tag{
    uint32_t hash[SHA256_HASH_SIZE/sizeof(uint32_t)];
    uint8_t  buffer[SHA256_BLOCK_SIZE];
    uint32_t totalBytes;
    uint8_t  bytes;
}sha256Context_t;
```

Description:

The context used by the SHA256 functions

Name:

```
typedef struct HMAC_SHA256_context_tag{
    sha256Context_t shaCtx;
    uint8_t pad[SHA256_BLOCK_SIZE];
}HMAC_SHA256_context_t;
```

Description:

The context used by the HMAC functions.

3.11.4 API Primitives

3.11.4.1 AES_128_Encrypt ()

Prototype:

```
void AES_128_Encrypt
(
uint8_t* pInput,
uint8_t* pKey,
uint8_t* pOutput
);
```

Description:

This function performs AES-128 encryption on a 16-byte block.

Parameters:

Name	Type	Direction	Description
pInput	uint8_t*	[IN]	Pointer to the location of the 16-byte plain text block.
pKey	uint8_t*	[IN]	Pointer to the location of the 128-bit key.
pOutput	uint8_t*	[OUT]	Pointer to the location to store the 16-byte ciphered output.

Returns:

None.

3.11.4.2 AES_128_Decrypt ()

Prototype:

```
void AES_128_Decrypt
(
uint8_t* pInput,
uint8_t* pKey,
uint8_t* pOutput
);
```

Description:
This function performs AES-128 decryption on a 16-byte block.

Parameters:

Name	Type	Direction	Description
pInput	uint8_t*	[IN]	Pointer to the location of the 16-byte ciphered text block.
pKey	uint8_t*	[IN]	Pointer to the location of the 128-bit key.
pOutput	uint8_t*	[OUT]	Pointer to the location to store the 16-byte plain text output.

Returns:

None.

3.11.4.3 AES_128_ECB_Encrypt ()

Prototype:

```
void AES_128_ECB_Encrypt
(
uint8_t* pInput,
uint32_t inputLen,
uint8_t* pKey,
uint8_t* pOutput
);
```

Description:
This function performs AES-128-ECB encryption on a message block.

Parameters:

Name	Type	Direction	Description
pInput	uint8_t*	[IN]	Pointer to the location of the input message.
inputLen	uint32_t	[IN]	Input message length in bytes.
pKey	uint8_t*	[IN]	Pointer to the location of the 128-bit key.
pOutput	uint8_t*	[OUT]	Pointer to the location to store the ciphered output.

Returns:

None.

3.11.4.4 AES_128_ECB_Block_Encrypt ()

Prototype:

```
void AES_128_ECB_Block_Encrypt
(
uint8_t* pInput,
uint32_t numBlocks,
uint8_t* pKey,
uint8_t* pOutput
);
```

Description:
This function performs AES-128-ECB encryption on a message block.

Parameters:

Name	Type	Direction	Description
pInput	uint8_t*	[IN]	Pointer to the location of the input message.
numBlocks	uint32_t	[IN]	Input message number of 16-byte blocks.
pKey	uint8_t*	[IN]	Pointer to the location of the 128-bit key.
pOutput	uint8_t*	[OUT]	Pointer to the location to store the ciphered output.

Returns:

None.

3.11.4.5 AES_128_CBC_Encrypt ()

Prototype:

```
void AES_128_CBC_Encrypt
(
uint8_t* pInput,
uint32_t inputLen,
uint8_t* pInitVector,
uint8_t* pKey,
uint8_t* pOutput
);
```

Description:
This function performs AES-128-CBC encryption on a message block.

Parameters:

Name	Type	Direction	Description
pInput	uint8_t*	[IN]	Pointer to the location of the input message.
inputLen	uint32_t	[IN]	Input message length in octets.
pInitVector	uint8_t*	[IN]	Pointer to the location of the 128-bit initialization vector.
pKey	uint8_t*	[IN]	Pointer to the location of the 128-bit key.
pOutput	uint8_t*	[OUT]	Pointer to the location to store the ciphered output.

Returns:

None.

3.11.4.6 AES_128_CTR ()

Prototype:

```
void AES_128_CTR
(
uint8_t* pInput,
uint32_t inputLen,
uint8_t* pCounter,
uint8_t* pKey,
uint8_t* pOutput
);
```

Description:
This function performs AES-128-CTR encryption on a message block.

Parameters:

Name	Type	Direction	Description
pInput	uint8_t*	[IN]	Pointer to the location of the input message.
inputLen	uint32_t	[IN]	Input message length in bytes.
pCounter	uint8_t*	[IN]	Pointer to the location of the 128-bit counter.
pKey	uint8_t*	[IN]	Pointer to the location of the 128-bit key.
pOutput	uint8_t*	[OUT]	Pointer to the location to store the ciphered output.

Returns:

None.

3.11.4.7 AES_128_OFB ()

Prototype:

```
void AES-128-OFB
(
uint8_t* pInput,
uint32_t inputLen,
uint8_t* pInitVector,
uint8_t* pKey,
uint8_t* pOutput
);
```

Description:
This function performs AES-128-OFB encryption on a message block.

Parameters:

Name	Type	Direction	Description
pInput	uint8_t*	[IN]	Pointer to the location of the input message.
inputLen	int32_t	[IN]	Input message length in bytes.
pInitVector	uint8_t*	[IN]	Pointer to the location of the 128-bit initialization vector.
pKey	uint8_t*	[IN]	Pointer to the location of the 128-bit key.
pOutput	uint8_t*	[OUT]	Pointer to the location to store the ciphered output.

Returns:

None.

3.11.4.8 AES_128_CMAC ()

Prototype:

```
void AES_128_CMAC
(
uint8_t* pInput,
uint32_t inputLen,
uint8_t* pKey,
uint8_t* pOutput
);
```

Description:
This function performs AES-128-CMAC on a message block.

Parameters:

Name	Type	Direction	Description
pInput	uint8_t*	[IN]	Pointer to the message block.
inputLen	uint32_t	[IN]	Length of the input message in bytes.
pKey	uint8_t*	[IN]	Pointer to the location of the 128-bit key.
pOutput	uint8_t*	[OUT]	Pointer to the location to store the 16-byte authentication code.

Returns:

None.

3.11.4.9 AES_128_EAX_Encrypt ()

Prototype:

```
typedef enum
{
    gSuccess_c,
    gSecurityError_c
```

```
} resultType_t;

resultType_t AES_128_EAX_Encrypt
(
uint8_t* pInput,
uint32_t inputLen,
uint8_t* pNonce,
uint32_t nonceLen,
uint8_t* pHeader,
uint8_t headerLen,
uint8_t* pKey,
uint8_t* pOutput
uint8_t* pTag
);
```

Description:

This function performs AES-128-EAX encryption on a message block.

Parameters:

Name	Type	Direction	Description
pInput	uint8_t*	[IN]	Pointer to the location of the input message.
inputLen	uint32_t	[IN]	Input message length in bytes.
pNonce	uint8_t*	[IN]	Pointer to the location of the nonce.
nonceLen	uint32_t	[IN]	Nonce length in bytes.
pHeader	uint8_t*	[IN]	Pointer to the location of header.
headerLen	uint32_t	[IN]	Header length in bytes.
pKey	uint8_t*	[IN]	Pointer to the location of the 128-bit key.
pOutput	uint8_t*	[OUT]	Pointer to the location to store the ciphered output.
pTag	uint8_t*	[OUT]	Pointer to the location to store the 128-bit tag.

Returns:

Operation status.

3.11.4.10 AES_128_EAX_Decrypt ()

Prototype:

```
typedef enum
{
    gSuccess_c,
    gSecurityError_c
} resultType_t;

resultType_t AES_128_EAX_Decrypt
(
uint8_t* pInput,
uint32_t inputLen,
uint8_t* pNonce,
uint32_t nonceLen,
uint8_t* pHeader,
uint8_t headerLen,
```

```
uint8_t* pKey,
uint8_t* pOutput
uint8_t* pTag
);
```

Description:
This function performs AES-128-EAX decryption on a message block.

Parameters:

Name	Type	Direction	Description
pInput	uint8_t*	[IN]	Pointer to the location of the input message.
inputLen	uint32_t	[IN]	Input message length in bytes.
pNonce	uint8_t*	[IN]	Pointer to the location of the nonce.
nonceLen	uint32_t	[IN]	Nonce length in bytes.
pHeader	uint8_t*	[IN]	Pointer to the location of header.
headerLen	uint32_t	[IN]	Header length in bytes.
pKey	uint8_t*	[IN]	Pointer to the location of the 128-bit key.
pOutput	uint8_t*	[OUT]	Pointer to the location to store the ciphered output.
pTag	uint8_t*	[OUT]	Pointer to the location to store the 128-bit tag.

Returns:

Operation status.

3.11.4.11 AES_128_CCM ()

Prototype:

```
void AES_128_CCM
(
   uint8_t* pInput,
   uint32_t inputLen,
   uint8_t* pAuthData,
   uint32_t authDataLen,
   uint8_t* pInitVector,
   uint8_t* pCounter,
   uint8_t* pKey,
   uint8_t* pOutput,
   uint8_t* pCbcMac,
   uint8_t flags
);
```

Description:
This function performs AES-128-CCM on a message block.

Parameters:

Name	Type	Direction	Description
pInput	uint8_t*	[IN]	Pointer to the location of the input message.
inputLen	uint32_t	[IN]	Input message length in bytes.

Name	Type	Direction	Description
pAuthData	uint8_t*	[IN]	Pointer to the additional authentication data
authDataLen	uint32_t	[IN]	The length of the additional authentication data.
pInitVector	uint8_t*	[IN]	Pointer to the location of the 128-bit initialization vector (B0).
pCounter	uint8_t*	[IN]	Pointer to the location of the 128-bit counter (A0).
pKey	uint8_t*	[IN]	Pointer to the location of the 128-bit key.
pOutput	uint8_t*	[OUT]	Pointer to the location to store the ciphered output.
pCbcMac	uint8_t*	[IN/OUT]	Encryption: pointer to the location to store the authentication code. Decription: pointer to the location of the received authentication code
flags	uint8_t	[IN]	Bit0 – 0 encrypt / 1 decrpt

Returns:

If the decrypt failed (MAC check failed) returns an error code. Else returns success.

3.11.4.12 SecLib_XorN ()

Prototype:

```
void SecLib_XorN
(
uint8_t* pDst,
uint8_t* pSrc,
uint8_t  len
);
```

Description:

This function performs XOR between pDst and pSrc, and stores the result at pDst.

Parameters:

Name	Type	Direction	Description
pDst	uint8_t*	[IN/OUT]	Pointer to the input/output data.
pSrc	uint8_t*	[IN]	Pointer to the input data.
len	uint8_t	[IN]	Data length in bytes

Returns:

None.

3.11.4.13 SHA1_Init ()

Prototype:

```
void SHA1_Init
```

```
(
sha1Context_t* context
);
```

Description:
This function performs SHA1 initialization.

Parameters:

Name	Type	Direction	Description
context	sha1Context_t*	[IN/OUT]	Pointer to the SHA1 context

Returns:

None.

3.11.4.14 SHA1_HashUpdate ()

Prototype:

```
void SHA1_HashUpdate
(
sha1Context_t* context,
uint8_t* pData,
uint32_t numBytes
);
```

Description:
This function performs SHA1 algorithm.

Parameters:

Name	Type	Direction	Description
context	sha1Context_t*	[IN/OUT]	Pointer to the SHA1 context
pData	uint8_t*	[IN]	Pointer to the plain text
numBytes	uint32_t	[IN]	Number of bytes

Returns:

None.

3.11.4.15 SHA1_HashFinish ()

Prototype:

```
void SHA1_HashFinish
(
sha1Context_t* context,
uint8_t* pData,
uint32_t numBytes
);
```

Description:
This function performs the final part of the SHA1 algorithm.

The final hash is stored into the context structure.

Parameters:

Name	Type	Direction	Description

context	sha1Context_t*	[IN/OUT]	Pointer to the SHA1 context
pData	uint8_t*	[IN]	Pointer to the plain text
numBytes	uint32_t	[IN]	Number of bytes

Returns:

None.

3.11.4.16 SHA1_Hash ()

Prototype:

```
void SHA1_Hash
(
sha1Context_t* context,
uint8_t* pData,
uint32_t numBytes
);
```

Description:
This function performs the entire SHA1 algorithm (initialize, update, finish) over the imput data.

The final hash is stored into the context structure.

Parameters:

Name	Type	Direction	Description
context	sha1Context_t*	[IN/OUT]	Pointer to the SHA1 context
pData	uint8_t*	[IN]	Pointer to the plain text
numBytes	uint32_t	[IN]	Number of bytes

Returns:

None.

3.11.4.17 SHA256_Init ()

Prototype:

```
void SHA256_Init
(
Sha256Context_t* context
);
```

Description:
This function performs SHA256 initialization.

Parameters:

Name	Type	Direction	Description
context	Sha256Context_t*	[IN/OUT]	Pointer to the SHA256 context

Returns:

None.

3.11.4.18 SHA256_HashUpdate ()

Prototype:

```
void SHA256_HashUpdate
(
Sha256Context_t* context,
uint8_t* pData,
uint32_t numBytes
);
```

Description:
This function performs SHA256 algorithm.

Parameters:

Name	Type	Direction	Description
Context	Sha256Context_t*	[IN/OUT]	Pointer to the SHA256 context
pData	uint8_t*	[IN]	Pointer to the plain text
numBytes	uint32_t	[IN]	Number of bytes

Returns:

None.

3.11.4.19 SHA256_HashFinish ()

Prototype:

```
void SHA256_HashFinish
(
Sha256Context_t* context,
uint8_t* pData,
uint32_t numBytes
);
```

Description:
This function performs the final part of the SHA256 algorithm.

The final hash is stored into the context structure.

Parameters:

Name	Type	Direction	Description
Context	Sha256Context_t*	[IN/OUT]	Pointer to the SHA256 context
pData	uint8_t*	[IN]	Pointer to the plain text
numBytes	uint32_t	[IN]	Number of bytes

Returns:

None.

3.11.4.20 SHA256_Hash ()

Prototype:

```
void SHA256_Hash
(
Sha256Context_t* context,
uint8_t* pData,
uint32_t numBytes
);
```

Description:
This function performs the entire SHA256 algorithm (initialize, update, finish) over the imput data.

The final hash is stored into the context structure.

Parameters:

Name	Type	Direction	Description
context	Sha256Context_t*	[IN/OUT]	Pointer to the SHA256 context
pData	uint8_t*	[IN]	Pointer to the plain text
numBytes	uint32_t	[IN]	Number of bytes

Returns:

None.

3.11.4.21 HMAC_SHA256_Init ()

Prototype:

```
void HMAC_SHA256_Init
(
HMAC_SHA256_context_t* ctx,
uint8_t* pKey,
uint32_t keyLen
);
```

Description:
This function performs HMAC initialization.

Parameters:

Name	Type	Direction	Description
ctx	HMAC_SHA256_context_t*	[IN/OUT]	Pointer to the HMAC context
pKey	uint8_t*	[IN]	Pointer to the HMAC key
keyLen	uint32_t	[IN]	Length of the key

Returns:

None.

3.11.4.22 HMAC_SHA256_Update ()

Prototype:

```
void HMAC_SHA256_Update
(
HMAC_SHA256_context_t* ctx,
uint8_t* pData,
uint32_t numBytes
);
```

Description:
This function performs HMAC algorithm based on SHA256.

Parameters:

Name	Type	Direction	Description
ctx	HMAC_SHA256_context_t*	[IN/OUT]	Pointer to the HMAC context
pData	uint8_t*	[IN]	Pointer to the plain text
numBytes	uint32_t	[IN]	Number of bytes

Returns:

None.

3.11.4.23 HMAC_SHA256_Finish ()

Prototype:

```
void HMAC_SHA256_Finish
(
HMAC_SHA256_context_t* ctx,
uint8_t* pData,
uint32_t numBytes
);
```

Description:
This function performs the final part of the HMAC algorithm.

The final hash is stored into the context structure.

Parameters:

Name	Type	Direction	Description
ctx	HMAC_SHA256_context_t	[IN/OUT]	Pointer to the HMAC context
pData	uint8_t*	[IN]	Pointer to the plain text
numBytes	uint32_t	[IN]	Number of block of bytes

Returns:

None.

3.11.4.24 HMAC_SHA256 ()

Prototype:

```
void HMAC_SHA256
(
HMAC_SHA256_context_t* ctx,
uint8_t* pKey,
uint32_t keyLen,
uint8_t* pMsg,
uint32_t msgLen
);
```

Description:

This function performs the entire HMAC algorithm (initialize, update, finish) over the imput data.

The final hash is stored into the context structure.

Parameters:

Name	Type	Direction	Description
context	HMAC_SHA256_context_t*	[IN/OUT]	Pointer to the HMAC context
pKey	uint8_t*	[IN]	Pointer to the HMAC key
keyLen	uint32_t	[IN]	Length of the key
pMsg	uint8_t*	[IN]	Pointer to the plain text
numBytes	uint32_t	[IN]	Number of block of bytes

Returns:

None.

3.12 Lists

3.12.1 Overview

The framework includes a general purpose linked lists module. It implements common lists operations:

- Get list from element
- Add to head
- Add to tail
- Remove head
- Get head
- Get next
- Get previous
- Remove element
- Add element before given element
- Get list size
- Get free places

3.12.2 User defined data type definitions

Name:

```
typedef enum
{
  gListOk_c = 0,
  gListFull_c,
  gListEmpty_c,
  gOrphanElement_c
}listStatus_t;
```

Description:

List status data type definition

Name:

```
typedef struct list_tag
{
  struct listElement_tag *head;
  struct listElement_tag *tail;
  uint16_t size;
  uint16_t max;
}list_t, *listHandle_t;
```

Description:

Data type definition for the list and list pointer.

Name:

```
typedef struct listElement_tag
{
  struct listElement_tag *next;
  struct listElement_tag *prev;
  struct list_tag *list;
}listElement_t, *listElementHandle_t;
```

Description:

Data type definition for the element and element pointer.

3.12.3 API Primitives

3.12.3.1 ListInit

Prototype:

```
void ListInit
(
    listHandle_t list,
    uint32_t max
);
```

Description:
Initializes the list descriptor.

Parameters:

Name	Type	Direction	Description
list	listHandle_t	[OUT]	Pointer to a list.
max	uint32_t	[IN]	Maximum number of elements in the list. 0 for unlimited.

Returns:

None

3.12.3.2 ListGetList

Prototype:

```
listHandle_t ListGetList
(
    listElementHandle_t elementHandle
);
```

Description:
Gets the list that contains the given element.

Parameters:

Name	Type	Direction	Description
elementHandle	listElementHandle_t	[IN]	Pointer to an element.

Returns:

Pointer to the list descriptor. Returns NULL if the element is orphan.

3.12.3.3 ListAddTail

Prototype:

```
listStatus_t ListAddTail
(
    listHandle_t list,
    listElementHandle_t element
```

```
);
```

Description:
Inserts an element at the end of the list.

Parameters:

Name	Type	Direction	Description
list	listHandle_t	[IN]	Pointer to a list.
element	listElementHandle_t	[IN]	Pointer to an element.

Returns:

gListFull_c if list is full. gListOk_c if insertion was successful.

3.12.3.4 ListAddHead

Prototype:

```
listStatus_t ListAddHead
(
    listHandle_t list,
    listElementHandle_t element
);
```

Description:
Inserts an element at the start of the list.

Parameters:

Name	Type	Direction	Description
list	listHandle_t	[IN]	Pointer to a list.
element	listElementHandle_t	[IN]	Pointer to an element.

Returns:

gListFull_c if list is full. gListOk_c if insertion was successful.

3.12.3.5 ListRemoveHead

Prototype:

```
listElementHandle_t ListRemoveHead
(
    listHandle_t list
);
```

Description:
Unlinks element from the head of the list.

Parameters:

Name	Type	Direction	Description
list	listHandle_t	[IN]	Pointer to a list.

Returns:

NULL if list is empty, pointer to the element if removal was successful.

3.12.3.6 ListGetHead

Prototype:

```
listElementHandle_t ListGetHead
(
    listHandle_t list
);
```

Description:
Gets the head element of the list.

Parameters:

Name	Type	Direction	Description
list	listHandle_t	[IN]	Pointer to a list.

Returns:

NULL if list is empty, pointer to the element if list is not empty.

3.12.3.7 ListGetNext

Prototype:

```
listElementHandle_t ListGetNext
(
    listElementHandle_t element
);
```

Description:
Gets the next element in list.

Parameters:

Name	Type	Direction	Description
element	listElementHandle_t	[IN]	Pointer to an element.

Returns:

NULL if given element is tail, pointer to the next element otherwise.

3.12.3.8 ListGetPrev

Prototype:

```
listElementHandle_t ListGetPrev
(
```

```
    listElementHandle_t element
);
```

Description:
Gets the previous element in list.

Parameters:

Name	Type	Direction	Description
element	listElementHandle_t	[IN]	Pointer to an element.

Returns:

NULL if given element is head, pointer to the previous element otherwise.

3.12.3.9 ListRemoveElement

Prototype:

```
listStatus_t ListRemoveElement
(
    listElementHandle_t element
);
```

Description:
Unlinks the given element from the list.

Parameters:

Name	Type	Direction	Description
element	listElementHandle_t	[IN]	Pointer to an element.

Returns:

gOrphanElement_c if element is not part of any list and gListOk_c if removal was successful.

3.12.3.10 ListAddPrevElement

Prototype:

```
listStatus_t ListAddPrevElement
(
    listElementHandle_t element,
    listElementHandle_t newElement
);
```

Description:
Links an element in the previous position relative to a given member of a list.

Parameters:

Name	Type	Direction	Description
element	listElementHandle_t	[IN]	Pointer to an element.
newElement	listElementHandle_t	[IN]	Pointer to the new element.

Returns:

gOrphanElement_c if element is not part of any list and gListOk_c if removal was successful.

3.12.3.11 ListGetSize

Prototype:

```
uint32_t ListGetSize
(
    listHandle_t list
);
```

Description:
Gets the current size of the list.

Parameters:

Name	Type	Direction	Description
list	listHandle_t	[IN]	Pointer to a list.

Returns:

Current size of the list.

3.12.3.12 ListGetAvailable

Prototype:

```
uint32_t ListGetSize
(
    listHandle_t list
);
```

Description:
Gets the number of free places in the list.

Parameters:

Name	Type	Direction	Description
list	listHandle_t	[IN]	Pointer to a list.

Returns:

Available spaces in the list.

3.12.4 Sample Code

Linked list example

```
typedef struct userStruct_tag
{
```

```
  listElement_t anchor;
  uint32_t data;
}userStruct_t;

list_t list;

uint32_t userFunc(void)
{
  userStruct_t dataStruct, *dataStruct_ptr;

  ListInit(list, 0);

  dataStruct.data = 56;

  ListAddTail(list, (listElementHandle_t)dataStruct.anchor);

  dataStruct_ptr = (userStruct_t *)ListRemoveHead(list);

  return dataStruct_ptr->data;
}
```

3.13 Function Lib

3.13.1 Overview

The framework provides a collection of features commonly used in embedded software centered on memory manipulation. Some features come in multiple flavors.

3.13.2 API Primitives

3.13.2.1 FLib_MemCpy, FLib_MemCpyAligned32bit, FLib_MemCpyDir

Prototype:

```
void FLib_MemCpy (void* pDst,     // IN: Pointer to destination memory block
                  void* pSrc,     // IN: Pointer to source memory block
                  uint32_t cBytes // IN: Number of bytes to copy
                  );


void FLib_MemCpyAligned32bit (void* to_ptr,
                              void* from_ptr,
                              register uint32_t number_of_bytes);


void FLib_MemCpyDir (void* pBuf1,
                     void* pBuf2,
                     bool_t dir,
                     uint32_t n);
```

Description:

Copy the content of one memory block to another.

Parameters:

Name	Type	Direction	Description

pDst to_ptr	void *	[OUT]	Pointer to the destination memory block.
pSrc from_ptr	void *	[IN]	Pointer to the source memory block.
cBytes number_of_bytes n	uint32_t	[IN]	Number of bytes to copy.
pBuf1 pBuf2	void *	[IN/OUT]	Pointer to a memory buffer
dir	bool_t	[IN]	Copying direction.

Returns:

None.

3.13.2.2 FLib_MemCpyReverseOrder

Prototype:

```
void FLib_MemCpyReverseOrder (void* pDst,      // Destination buffer
                              void* pSrc,      // Source buffer
                              uint32_t cBytes  // Byte count
                             );
```

Description:
Copies the byte at index i from the source buffer is copied to index ((n-1) - i) in the destination buffer (and vice versa).

Parameters:

Name	Type	Direction	Description
pDst	void *	[OUT]	Pointer to the destination memory block.
pSrc	void *	[IN]	Pointer to the source memory block.
cBytes	uint32_t	[IN]	Number of bytes to copy.

Returns:

None.

3.13.2.3 FLib_MemCmp

Prototype:

```
bool_t FLib_MemCmp (void* pData1,   // IN: First memory block to compare
                    void* pData2,   // IN: Second memory block to compare
                    uint32_t cBytes // IN: Number of bytes to compare.
                   );
```

Description:
Compare two memory blocks.

Parameters:

Name	Type	Direction	Description

pData1	void *	[IN]	Pointer to a memory block.
pData2	void *	[IN]	Pointer to a memory block.
cBytes	uint32_t	[IN]	Number of bytes to copy.

Returns:

If the blocks are equal byte by byte, the function returns TRUE, and FALSE otherwise.

3.13.2.4 FLib_MemSet, FLib_MemSet16

Prototype:

```
void FLib_MemSet (void* pData,     // IN: Pointer to memory block to reset
                  uint8_t value,   // IN: Value that memory block will be reset to.
                  uint32_t cBytes  // IN: Number of bytes to reset.
                 );
void FLib_MemSet16 (void* pDst,        // Buffer to be reset
                    uint8_t value,     // Byte value
                    uint32_t cBytes    // Byte count
                   );
```

Description:
Reset bytes in a memory block to a certain value. One function operates on 8 bit aligned blocks, the other on 16 bit aligned blocks.

Parameters:

Name	Type	Direction	Description
pData	void *	[OUT]	Pointer to a memory block.
value	uint8_t	[IN]	Value.
cBytes	uint32_t	[IN]	Number of bytes to reset.

Returns:

None.

3.13.2.5 FLib_MemInPlaceCpy

Prototype:

```
void FLib_MemInPlaceCpy (void* pDst,      // Destination buffer
                         void* pSrc,      // Source buffer
                         uint32_t cBytes  // Byte count
                        );
```

Description:
Copies bytes, possibly into the same overlapping memory as it is taken from.

Parameters:

Name	Type	Direction	Description

pDst	void *	[OUT]	Pointer to the destination memory block.
pSrc	void *	[IN]	Pointer to the source memory block.
cBytes	uint32_t	[IN]	Number of bytes to reset.

Returns:

None.

3.13.2.6 FLib_MemCopy16Unaligned, FLib_MemCopy32Unaligned, FLib_MemCopy64Unaligned

Prototype:

```
void FLib_MemCopy16Unaligned (void* pDst,      // Pointer to destination memory block
                              uint16_t val16   // The value to be copied
                             );
void FLib_MemCopy32Unaligned (void* pDst,      // Pointer to destination memory block
                              uint32_t val32   // The value to be copied
                             );
void FLib_MemCopy64Unaligned (void* pDst,      // Pointer to destination memory block
                              uint64_t val64   // The value to be copied
                             );
```

Description:

Copies a 16, 32 and 64 bit value to an unaligned a memory block.

Parameters:

Name	Type	Direction	Description
pDst	void *	[OUT]	Pointer to the destination memory block.
val16 val32 val64	uint16_t uint32_t uint64_t	[IN]	Value to set the buffer to.

Returns:

None.

3.13.2.7 FLib_AddOffsetToPointer

Prototype:

```
void FLib_AddOffsetToPointer (void** pPtr, uint32_t offset);
#define FLib_AddOffsetToPtr(pPtr,offset) FLib_AddOffsetToPointer((void**)(pPtr),(offset))
```

Description:

Adds an offset to a pointer.

Parameters:

Name	Type	Direction	Description

pPtr	void **	[OUT]	Pointer to add offset to.
offset	uint32_t	[IN]	Offset value.

Returns:

None.

3.13.2.8 FLib_Cmp2Bytes

Prototype:

```
#define FLib_Cmp2Bytes(c1, c2) (*((uint16_t*) c1) == *((uint16_t*) c2))
```

Description:
Compares two bytes.

Parameters:

Name	Type	Direction	Description
c1	-	[IN]	Value.
c2	-	[IN]	Value.

Returns:

TRUE if content of buffers is equal, and FALSE otherwise.

3.13.2.9 FLib_GetMax

Prototype:

```
#define FLib_GetMax(a,b)    (((a) > (b)) ? (a) : (b))
```

Description:
Returns the maximum value of arguments a and b.

Parameters:

Name	Type	Direction	Description
a	-	[IN]	Value.
b	-	[IN]	Value.

Returns:

The maximum value of arguments a and b.

3.13.2.10 FLib_GetMin

Prototype:

```
#define FLib_GetMin(a,b)    (((a) < (b)) ? (a) : (b))
```

Description:
Returns the minimum value of arguments a and b.

Parameters:

Name	Type	Direction	Description
a	-	[IN]	Value.
b	-	[IN]	Value.

Returns:

The minimum value of arguments a and b.

4 Drivers

4.1.1 Overview

A set of high level drivers for LEDs and keyboard is provided but are not part of the framework. The drivers require the Timers Manager module to be included in the project.

4.2 LED

4.2.1 Overview

The module allows control of up to 4 LEDs using a low level driver. It offers a high level API for various operation modes:

- Flashing
- Serial flashing
- Blip
- Solid on
- Solid off
- Toggle

The flash and blip features use a timer from the Timers Manager module.

4.2.2 Constant Macro Definitions

Name:

```
#define gLEDSupported_d         TRUE
```

Description:

Enables/disables the LED module.

Name:

```
#define gLEDsOnTargetBoardCnt_c 4
```

Description:

Configures the number of LEDs used up to a maximum of 4.

Name:

```
#define gLEDBlipEnabled_d       TRUE
```

Description:

Enables/disables the blip feature.

Name:

```
#define mLEDInterval_c          100
```

Description:

Configures the on period of the flashing feature in milliseconds.

Name:

```
#define LED1                    0x01
#define LED2                    0x02
#define LED3                    0x04
#define LED4                    0x08
#define LED_ALL                 0x0F
```

Description:

LEDs mapping.

4.2.3 User defined data type definitions

Name:

```
typedef uint8_t LED_t;
```

Description:

LED type definition.

Name:

```
typedef enum LED_OpMode_tag{
    gLedFlashing_c,       /* flash at a fixed rate */
    gLedStopFlashing_c,   /* same as gLedOff_c */
    gLedBlip_c,           /* just like flashing, but blinks only once */
    gLedOn_c,             /* on solid */
    gLedOff_c,            /* off solid */
    gLedToggle_c          /* toggle state */
} LED_OpMode_t;
```

Description:

Enumerated data type for all possible LED operation modes.

Name:

```
typedef uint8_t LedState_t;
```

Description:

Possible LED states for LED_SetLed().

4.2.4 API Primitives

4.2.4.1 TurnOnLeds ()

Prototype:

```
#define TurnOnLeds()            LED_TurnOnAllLeds()
```

Description:
Turns on all LEDs.

Parameters:

None.

Returns:

None.

4.2.4.2 SerialFlasing ()

Prototype:

```
#define SerialFlashing()        LED_StartSerialFlash()
```

Description:
Turns on serial flashing on all LEDs.

Parameters:

None.

Returns:

None.

4.2.4.3 Led1Flashing (), Led2Flashing (), Led3Flashing (), Led4Flashing ()

Prototype:

```
#define Led1Flashing()          LED_StartFlash(LED1)
#define Led2Flashing()          LED_StartFlash(LED2)
#define Led3Flashing()          LED_StartFlash(LED3)
#define Led4Flashing()          LED_StartFlash(LED4)
```

Description:
Turns flashing on for each LED.

Parameters:

None.

Returns:

None.

4.2.4.4 StopLed1Flashing(), StopLed2Flashing(), StopLed3Flashing(), StopLed4Flashing()

Prototype:

```
#define StopLed1Flashing()       LED_StopFlash(LED1)
#define StopLed2Flashing()       LED_StopFlash(LED2)
#define StopLed3Flashing()       LED_StopFlash(LED3)
#define StopLed4Flashing()       LED_StopFlash(LED4)
```

Description:
Turns flashing off for each LED.

Parameters:

None.

Returns:

None.

4.2.4.5 LED_Init ()

Prototype:

```
extern void LED_Init
(
        void
);
```

Description:
Initializes the LED module.

Parameters:

None.

Returns:

None.

4.2.4.6 LED_UnInit ()

Prototype:

```
extern void LED_Init
(
        void
);
```

Description:
Turns off all the LEDs and disables clock gating for LED port.

Parameters:

None.

Returns:

None.

4.2.4.7 LED_Operate ()

Prototype:

```
extern void LED_Operate
(
        LED_t led,
        LED_OpMode_t operation
);
```

Description:
Basic LED operation: ON, OFF, TOGGLE.

Parameters:

Name	Type	Direction	Description
led	LED_t	[IN]	LED(s) to operate.
operation	LED_OpMode_t	[IN]	LED operation.

Returns:

None.

4.2.4.8 LED_TurnOnLed ()

Prototype:

```
extern void LED_TurnOnLed
(
        LED_t LEDNr
);
```

Description:

Turns ON the specified LED(s).

Parameters:

Name	Type	Direction	Description
LEDNr	LED_t	[IN]	LED(s) to operate.

Returns:

None.

4.2.4.9 LED_TurnOffLed ()

Prototype:

```
extern void LED_TurnOffLed
(
        LED_t LEDNr
);
```

Description:

Turns OFF the specified LED(s).

Parameters:

Name	Type	Direction	Description
LEDNr	LED_t	[IN]	LED(s) to operate.

Returns:

None.

4.2.4.10 LED_ToggleLed ()

Prototype:

```
extern void LED_ToggleLed
(
        LED_t LEDNr
);
```

Description:

Toggles the specified LED(s).

Parameters:

Name	Type	Direction	Description
LEDNr	LED_t	[IN]	LED(s) to operate.

Returns:

None.

4.2.4.11 LED_TurnOffAllLeds ()

Prototype:

```
extern void LED_TurnOffAllLeds
(
        void
);
```

Description:
Turns off all the LEDs.

Parameters:

None.

Returns:

None.

4.2.4.12 LED_TurnOnAllLeds ()

Prototype:

```
extern void LED_TurnOnAllLeds
(
        void
);
```

Description:
Turns on all the LEDs.

Parameters:

None.

Returns:

None.

4.2.4.13 LED_StopFlashingAllLeds ()

Prototype:

```
extern void LED_StopFlashingAllLeds
(
        void
);
```

Description:
Stops flashing and turns OFF all LEDs.

Parameters:

None.

Returns:

None.

4.2.4.14 LED_StartFlash ()

Prototype:

```
void LED_StartFlash
(
        LED_t LEDNr
);
```

Description:

Starts flashing one or more LEDs.

Parameters:

Name	Type	Direction	Description
LEDNr	LED_t	[IN]	LED(s) to operate.

Returns:

None.

4.2.4.15 LED_StartBlip ()

Prototype:

```
extern void LED_StartBlip
(
        LED_t LEDNr
);
```

Description:

Set up for blinking one or more LEDs once.

Parameters:

Name	Type	Direction	Description
LEDNr	LED_t	[IN]	LED(s) to operate.

Returns:

None.

4.2.4.16 LED_StopFlash ()

Prototype:

```
extern void LED_StopFlash
(
        LED_t LEDNr
);
```

Description:

Stop an LED from flashing.

Parameters:

Name	Type	Direction	Description
LEDNr	LED_t	[IN]	LED(s) to operate.

Returns:

None.

4.2.4.17 LED_StartSerialFlash ()

Prototype:

```
extern void LED_StartSerialFlash
(
        void
);
```

Description:

Starts serial flashing LEDs.

Parameters:

None.

Returns:

None.

4.2.4.18 LED_SetHex ()

Prototype:

```
extern void LED_SetHex
(
        uint8_t hexValue
);
```

Description:

Sets a specified hex value on the LEDs.

Parameters:

Name	Type	Direction	Description
hexValue	uint8_t	[IN]	Hex value.

Returns:

None.

4.2.4.19 LED_SetLed ()

Prototype:

```
extern void LED_SetLed
(
        LED_t LEDNr,
        LedState_t state
);
```

Description:

Sets a specified hex value on the LEDs.

Parameters:

Name	Type	Direction	Description
LEDNr	LED_t	[IN]	LED(s) to operate.
state	LedState_t	[IN]	State to put the LED(s) into.

Returns:

None.

4.3 Keyboard

4.3.1 Overview

The module allows control of up to 4 switches using a low level driver. It offers a high level API for various operation modes:

- Press only
- Short/long press
- Press/hold/release

The keyboard event is received by the user code by executing a user implemented callback in interrupt context. The keyboard uses a timer from the Timers Manager module for debouncing, short/log press and hold detection.

4.3.2 Constant Macro Definitions

Name:

```
#define gKeyBoardSupported_d        TRUE
```

Description:

Enables/disables the keyboard module.

Name:

```
#define gKBD_KeysCount_c            4
```

Description:

Configures the number of switches.

Name:

```
#define gKeyEventNotificationMode_d gKbdEventShortLongPressMode_c
```

Description:

Selects the operation mode of the keyboard module.

Name:

```
#define gKbdEventPressOnly_c                1
#define gKbdEventShortLongPressMode_c       2
#define gKbdEventPressHoldReleaseMode_c     3
```

Description:

Mapping for keyboard operation modes.

Name:

```
#define gKbdLongKeyIterations_c             20
```

Description:

The iterations required for key long press detection. The detection threshold is gKbdLongKeyIterations_c x gKeyScanInterval_c milliseconds.

Name:

```
#define gKbdFirstHoldDetectIterations_c     20 /* 1 second, if gKeyScanInterval_c = 50ms */
```

Description:

The iterations required for key hold detection.

Name:

```
#define gKbdHoldDetectIterations_c          20 /* 1 second, if gKeyScanInterval_c = 50ms */
```

Description:

The iterations required for key hold detection (repetitive generation of event). May be the same value as gKbdFirstHoldDetectIterations_c.

Name:

```
#define gKeyScanInterval_c                  50 /* default is 50 milliseconds */
```

Description:

Constant for a key press. A short key will be returned after this number of millisecond if pressed make sure this constant is long enough for debounce time.

4.3.3 User defined data type definitions

Name:

```
typedef void (*KBDFunction_t) ( uint8_t events );
```

Description:

Callback function type definition.

Name:

```
typedef uint8_t key_event_t;
```

Description:

Each key delivered to the callback function is of this type (see the following enumerations).

Name:

```
enum
{
    gKBD_EventPB1_c = 1,          /* Pushbutton 1 */
    gKBD_EventPB2_c,              /* Pushbutton 2 */
    gKBD_EventPB3_c,              /* Pushbutton 3 */
    gKBD_EventPB4_c,              /* Pushbutton 4 */
    gKBD_EventLongPB1_c,          /* Pushbutton 1 */
    gKBD_EventLongPB2_c,          /* Pushbutton 2 */
    gKBD_EventLongPB3_c,          /* Pushbutton 3 */
    gKBD_EventLongPB4_c,          /* Pushbutton 4 */
};
```

Description:

Key code that is given to the callback function.

Name:

```
enum
{
    gKBD_EventPressPB1_c = 1,
    gKBD_EventPressPB2_c,
    gKBD_EventPressPB3_c,
    gKBD_EventPressPB4_c,
    gKBD_EventHoldPB1_c,
    gKBD_EventHoldPB2_c,
    gKBD_EventHoldPB3_c,
    gKBD_EventHoldPB4_c,
    gKBD_EventReleasePB1_c,
```

```
    gKBD_EventReleasePB2_c,
    gKBD_EventReleasePB3_c,
    gKBD_EventReleasePB4_c,
};
```

Description:

Key code that is given to the callback function.

Name:

```
#define gKBD_EventSW1_c                 gKBD_EventPB1_c
#define gKBD_EventLongSW1_c             gKBD_EventLongPB1_c
#define gKBD_EventSW2_c                 gKBD_EventPB2_c
#define gKBD_EventLongSW2_c             gKBD_EventLongPB2_c
#define gKBD_EventSW3_c                 gKBD_EventPB3_c
#define gKBD_EventLongSW3_c             gKBD_EventLongPB3_c
#define gKBD_EventSW4_c                 gKBD_EventPB4_c
#define gKBD_EventLongSW4_c             gKBD_EventLongPB4_c
```

Description:

Short/long press mode event mapping.

Name:

```
#define gKBD_EventPressSW1_c            gKBD_EventPressPB1_c
#define gKBD_EventHoldSW1_c             gKBD_EventHoldPB1_c
#define gKBD_EventReleaseSW1_c          gKBD_EventReleasePB1_c
#define gKBD_EventPressSW2_c            gKBD_EventPressPB2_c
#define gKBD_EventHoldSW2_c             gKBD_EventHoldPB2_c
#define gKBD_EventReleaseSW2_c          gKBD_EventReleasePB2_c
#define gKBD_EventPressSW3_c            gKBD_EventPressPB3_c
#define gKBD_EventHoldSW3_c             gKBD_EventHoldPB3_c
#define gKBD_EventReleaseSW3_c          gKBD_EventReleasePB3_c
#define gKBD_EventPressSW4_c            gKBD_EventPressPB4_c
#define gKBD_EventHoldSW4_c             gKBD_EventHoldPB4_c
#define gKBD_EventReleaseSW4_c          gKBD_EventReleasePB4_c
```

Description:

Press/hold/release mode event mapping.

4.3.4 API Primitives

4.3.4.1 KBD_Init ()

Prototype:

```
extern void KBD_Init
(
        KBDFunction_t pfCallBackAdr
);
```

Description:

Initializes the keyboard module internal variables.

Parameters:

Name	Type	Direction	Description
pfCallBackAdr	KBDFunction_t	[IN]	Pointer to application callback function.

Returns:

None.

4.3.4.2 KBD_IsWakeupSource

Prototype:

```
bool_t KBD_IsWakeUpSource
(
  void
);
```

Description:
Indicates if a keyboard event triggered a CPU wake-up.

Parameters:

None.

Returns:

TRUE if the keyboard was the wake-up source and FALSE otherwise.

4.4 GPIO Irq Adapter

4.4.1 Overview

The ARM core allows installation of an ISR for an entire GPIO port. The module allows the installation of a callback functions for one or more GPIO pins, with different priorities.

The module installs a common ISR for all MCU PORTs, and handles the installed callbacks based on the priority level set.

4.4.2 Constant Macro Definitions

Name:

```
#define gGpioMaxIsrEntries_c   (5)
```

Description:

Configures the maximum numner of entries in the GPIO ISR callback table.

Name:

```
#define gGpioIsrPrioHigh_c     (0)
#define gGpioIsrPrioNormal_c   (7)
#define gGpioIsrPrioLow_c      (15)
```

Description:

Defines basic priority levels to be used when registering ISR callbacks.

4.4.3 Data type definitions

Name:

```
typedef void (*pfGpioIsrCb_t)(void);
```

Description:

The GPIO ISR callback type

Name:

```
typedef struct gpioIsr_tag{
    pfGpioIsrCb_t callback;
    uint32_t      pinMask;
    IRQn_Type     irqId;
    uint8_t       port;
    uint8_t       prio;
}gpioIsr_t;
```

Description:

Defines an entry of the GPIO ISR table

Name:

```
typedef enum gpioStatus_tag{
    gpio_success,
    gpio_outOfMemory,
    gpio_notFound,
    gpio_error
}gpioStatus_t;
```

Descriprion:

Defines the error codes returned by the API

4.4.4 API Primitives

4.4.4.1 GpioInstallIsr

Prototype:

```
gpioStatus_t GpioInstallIsr
(
pfGpioIsrCb_t cb,
uint8_t priority,
uint8_t nvicPriority,
uint32_t pinDef
);
```

Description:
Installs an ISR callback for the specifier GPIO

Parameters:

Name	Type	Direction	Description
cb	pfGpioIsrCb_t	[IN]	Pointer to application callback function.
priority	uint8_t	[IN]	The priority of the callback
nvicPriority	uint8_t	[IN]	The priority to be set in NVIC
pinDef	Uint32_t	[IN]	KSDK PIN definition

Returns:

The error code .

4.4.4.2 GpioUninstallIsr

Prototype:

```
gpioStatus_t GpioUninstallIsr
(
uint32_t pinDef
);
```

Description:
Uninstalls the ISR callback for the specifier GPIO

Parameters:

Name	Type	Direction	Description
pinDef	Uint32_t	[IN]	KSDK PIN definition

Returns:

The error code.

Table 3 Revision history

Revision	Substantial changes
0	Initial release.

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support4

Document number: CONNFWKRM
Rev. 4
03/2015