Freescale Semiconductor, Inc.
Reference Manual

Document Number: CONNFWKRM
Rev. 1, 09/2015

Connectivity Framework

1. Introduction

The scope of this document is the Connectivity Framework software used to ensure portability across the ARM®-based MCU 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

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
LPM	Low Power Module

2. Overview

The system architecture decomposition is depicted in the following figure. As you can see, the framework, FSCI (Test Client), and the components are at same level, offering their services to the upper layers.

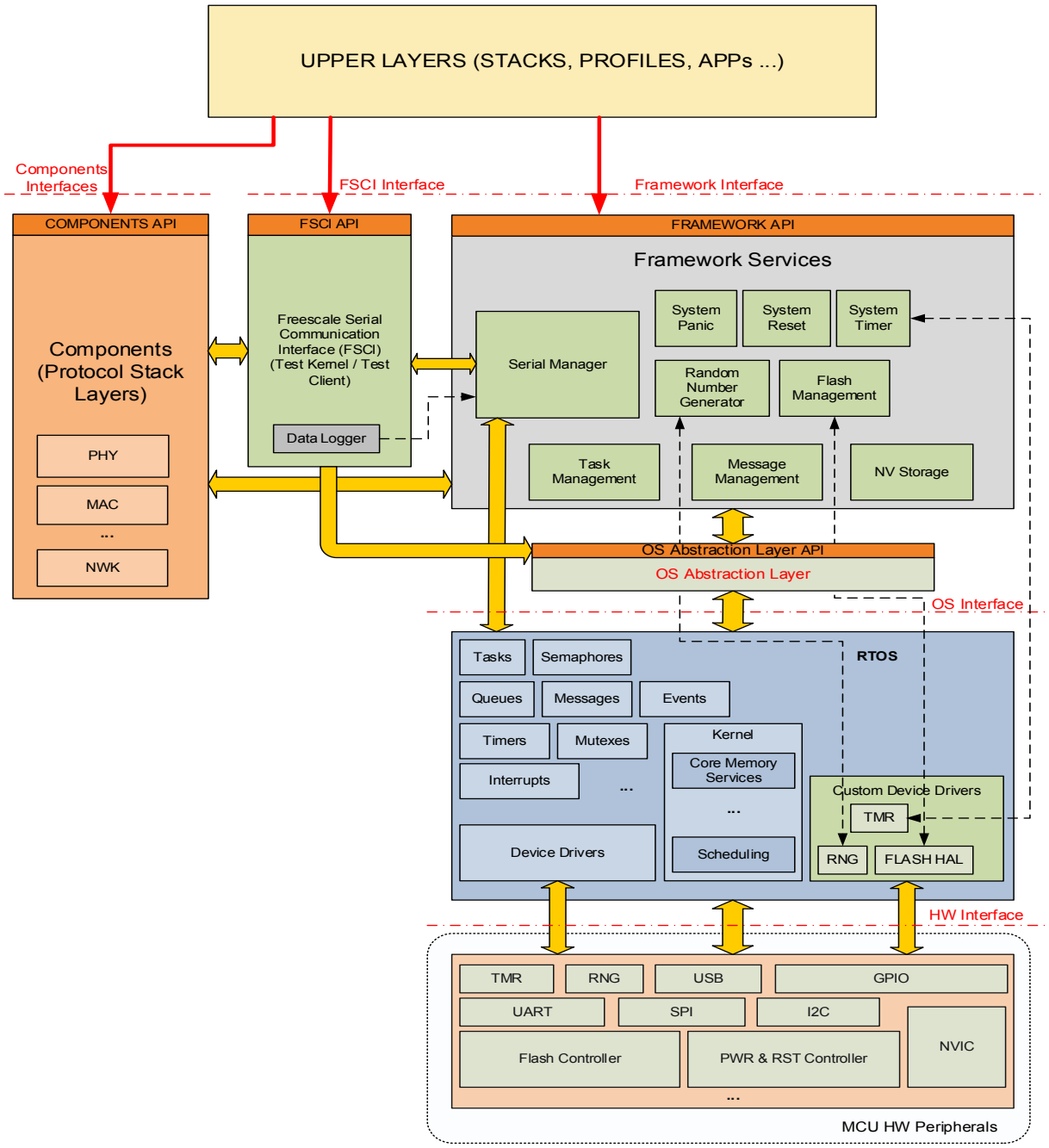


Figure 1. System decomposition

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

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 that 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 much different, for example the implementation of signals and timers.

To add support for another operating system, all functions and macros detailed in this section must be implemented for each RTOS. In addition, all typedefs must be analyzed and modified if needed.

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, and so on, restrictions and enhancements are needed within the OS Abstraction layer that reflects throughout the code.

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 RTOS services and bare-metal environment. 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, the task named main_task() is used as the starting point. You 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, you 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, because the task stack is a global array.

The *main_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 the 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 you want to, using the *OSA_EXT_TaskCreate* API.

For MQX, task stacks are arrays defined by the *_EXT_TASK_DEFINE* macro, and for FreeRTOS the 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 note that for MQX, the task stack cannot be freed, because 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:

Defines a task using the name as an identifier.

- priority – the task priority
- instances – the maximum number of instances the task can have
- stackSz – the task stack size in bytes
- useFloat – specifies, whether the task uses float operations or not

Name:

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

Description:

Used to reference a thread definition by name.

Name:

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

Description:

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, adds it to active threads, and sets it to the READY state.

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 the READY state.

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 cannot 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 for.

Returns:

None.

Example 1. 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 for 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 cannot 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 cannot be signaled, or an 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 frees 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 cannot be destroyed.

OSA_EXT_MutexLock ()

Prototype:

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

Description:
Takes a 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 an 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, the queue allocation is moved into the OS Abstraction layer. In both operating systems, passing messages through the queues is done by copy and not by reference. Messages are defined to be single 32-bit values or pointers.

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 into 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 is successfully put into the queue.
- osaStatus_Error – The queue is full, or an invalid parameter is 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 – The message is successfully obtained from the queue.
- osaStatus_Timeout – The queue remains empty after timeout.
- osaStatus_Error – 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 waits for any signal flag. Else, it waits 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:
Sets the specified signal flags of an active thread.

Parameters:

Name	Type	Direction	Description
autoClear	Bool_t	[IN]	If TRUE, event flags are 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:

Sets 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:

Sets 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:
Clears 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 waits 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 been set

Returns:

- osaStatus_Success – The wait condition is met and function returns successfully.
- osaStatus_Timeout – The wait condition was not met within the 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 waits for any signal flag. Else, it waits 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 successful, 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:
Stops 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 deallocates 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 an 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
- Returns NULL if the handler cannot be installed.

3.2. Message management

3.2.1. Overview

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

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. You must allocate a message buffer, and 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, you can use semaphores, mutexes, signals, and so on. It is recommended to use the existing signals, since they are created for every task, to avoid consuming extra memory.

You can send a message to a receiving task and then activate the synchronization element, and on the other side, it first waits for the synchronization signal, and then dequeues the message. The actual memory buffer is allocated by the sending task, and it 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

MSG_Queue

Prototype:

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

Description:
Puts a message into 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.

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.

MSG_Pending

Prototype:

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

Description:

Checks if a message is pending in the queue.

Parameters:

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

Returns:

Returns TRUE if there are any pending messages, and FALSE otherwise.

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.

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.

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 by the Memory manager.

Returns:

Pointer to the newly allocated block, or NULL if failed.

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.

MSG_Free

Prototype:

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

Description:
Frees a message.

Parameters:

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

Returns:

Status of the freeing operation.

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.

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.

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.

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.

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

Due to the memory fragmentation issues that usually appear, a generic allocation scheme is not desirable to be used, 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 returns the first available partition block of equal or higher size (that is, if no buffer of the requested size is available, the allocation routine returns a buffer of a larger size). As per requirements, the partition must be defined in ascending size order, and the block sizes must be multiples of four, to ensure the block alignment to four bytes:

Example 2. Defines pools by block size and number of blocks. Must be aligned to four 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 must be created. In addition to the requested amount of memory, each block will use a header for internal bookkeeping. You must not use this header, 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 the statuses used in MEM_BufferAlloc and MEM_BufferFree.

3.3.3. Memory management API primitives

MEM_Init()

Prototype:

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

Description:

This function is used to initialize the memory management subsystem. The function allocates memory for all partitions; it 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.

MEM_BufferAlloc()

Prototype:

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

Description:

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

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

Call MEM_BufferFree() only on a pointer that was returned by MEM_BufferAlloc(). Do not call the free function for a buffer that is already free.

MEM_GetAvailableBlocks()

Prototype:

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

Description:

This 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 whose basis the buffers are queried

Returns:

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

MEM_BufferGetSize()

Prototype:

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

Description:

This 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.

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 successufuly, `MEM_FREE_ERROR_c` if a buffer was not freed successufuly, or `MEM_UNKNOWN_ERROR_c` if a verify error, heap overflow or data corruption occurred.

3.3.4. Sample code

Example 3. 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, when compared to the OS-based timing functions. The Timers Manager operates at the peripheral clock frequency, while the OS timer frequency is set to 200 Hz (5 ms period).

The following services are provided by the Timers Manager:

- Initialize a module
- 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 times out

There are two types of timers provided:

- Single Shot Timers – run only once until they time out. They can be stopped before the timeout.
- Interval Timers – run continuously, and time out at the set regular intervals until explicitly stopped.

Each allocated timer has an associated callback function that is called from the interrupt execution context, and, therefore, it 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 is actually greater than or equal to 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 times out, so the mechanism is more efficient when compared to the OS-managed timing, which requires the execution of periodic (default is 5 ms) interrupts.

Timers can be identified as low-power timers on creation. This means that low-power timers do not run in low-power modes, rather they are synchronized when exiting the low-power mode. If a low-power timer expires when the MCU sleeps, its expiration is processed when the MCU exits sleep.

If hardware low-power timers are enabled (*gTMR_EnableHWLowPowerTimers_d = 1*), then the MCU exits 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 must not 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 μs 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 timer module requires its own initialization.

3.4.2. Constant macro definitions

Name:

```
#define gTMR_Enabled_d                    TRUE
```

Description:

Enables / disables the timer module, except for the 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 hardware 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 exiting the low-power mode.

Name:

```
#define gTMR_EnableHWLowPowerTimers_d        FALSE
```

Description:

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

TMR_Init()

Prototype:

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

Description:

This function initializes the system timer module, and it 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.

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 allocated first. After the timer is allocated, its internal status is set to inactive.

Parameters:

None.

Returns:

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

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.

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 set up 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 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.

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 doesn't do anything. Stopping a timer doesn't automatically free it. After it is stopped, the specified timer timeout events are deactivated until the timer is restarted.

Parameters:

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

Returns:

None.

TMR_IsTimerActive

Prototype:

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

Description:

This function checks whether 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.

TMR_GetRemainingTime

Prototype:

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

Description:

This function returns the time (in milliseconds) until the specified timer expires (times out), or 0 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.

TMR_EnableTimer

Prototype:

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

Description:

This 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.

TMR_AreAllTimersOff

Prototype:

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

Description:

Checks 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.

TMR_IsTimerReady

Prototype:

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

Description:

Checks 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.

TMR_StartLowPowerTimer

Prototype:

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

Description:

Starts 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 the low-power mode 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.

TMR_StartMinuteTimer

Prototype:

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

Description:

Starts 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.

TMR_StartSecondTimer

Prototype:

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

Description:

Starts 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.

TMR_StartIntervalTimer

Prototype:

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

Description:

Starts 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.

TMR_StartSingleShotTimer

Prototype:

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

Description:

Starts 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.

TMR_TimestampInit()

Prototype:

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

Description:

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

Parameters:

None.

Returns:

None.

TMR_GetTimestamp()

Prototype:

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

Description:

Returns the absolute time at the moment of the call.

Parameters:

None.

Returns:

Timestamp (in μs).

TMR_RTCInit()

Prototype:

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

Description:

The function initializes the RTC HW.

Parameters:

None.

Returns:

None.

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 μs).

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.

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.

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 nonvolatile data. Flash memories have individually eraseable segments (sectors), and each segment has a limited number of erase cycles. If the same segments are used all the time to store different kinds of data, those segments will become unreliable in a short time. Therefore, a wear-leveling mechanism is necessary 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().

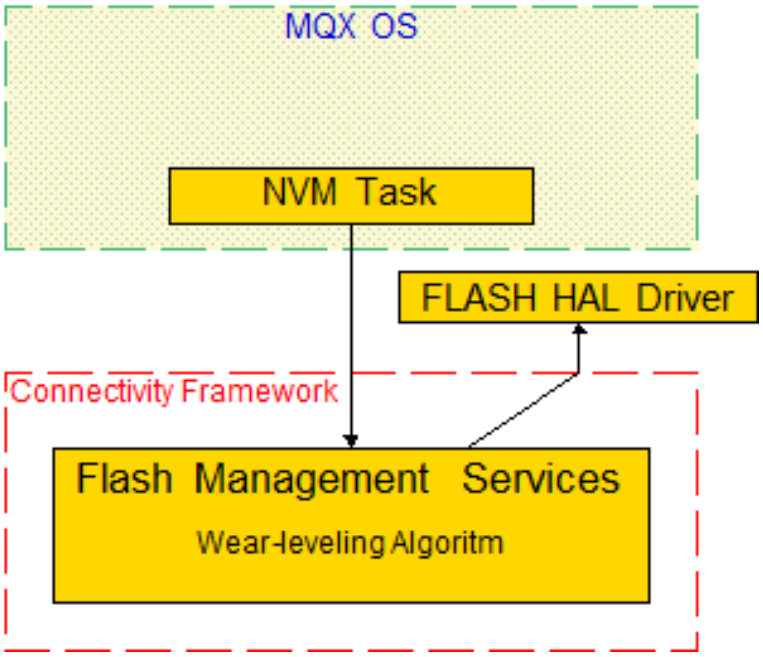


Figure 2. NVM overview

3.5.2. Standard storage system

Most of the MCUs have only a standard flash memory that can be used by the nonvolatile storage system. The amount of memory that the system uses 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 register information about the data that must 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, a table entry ID and an entry type.

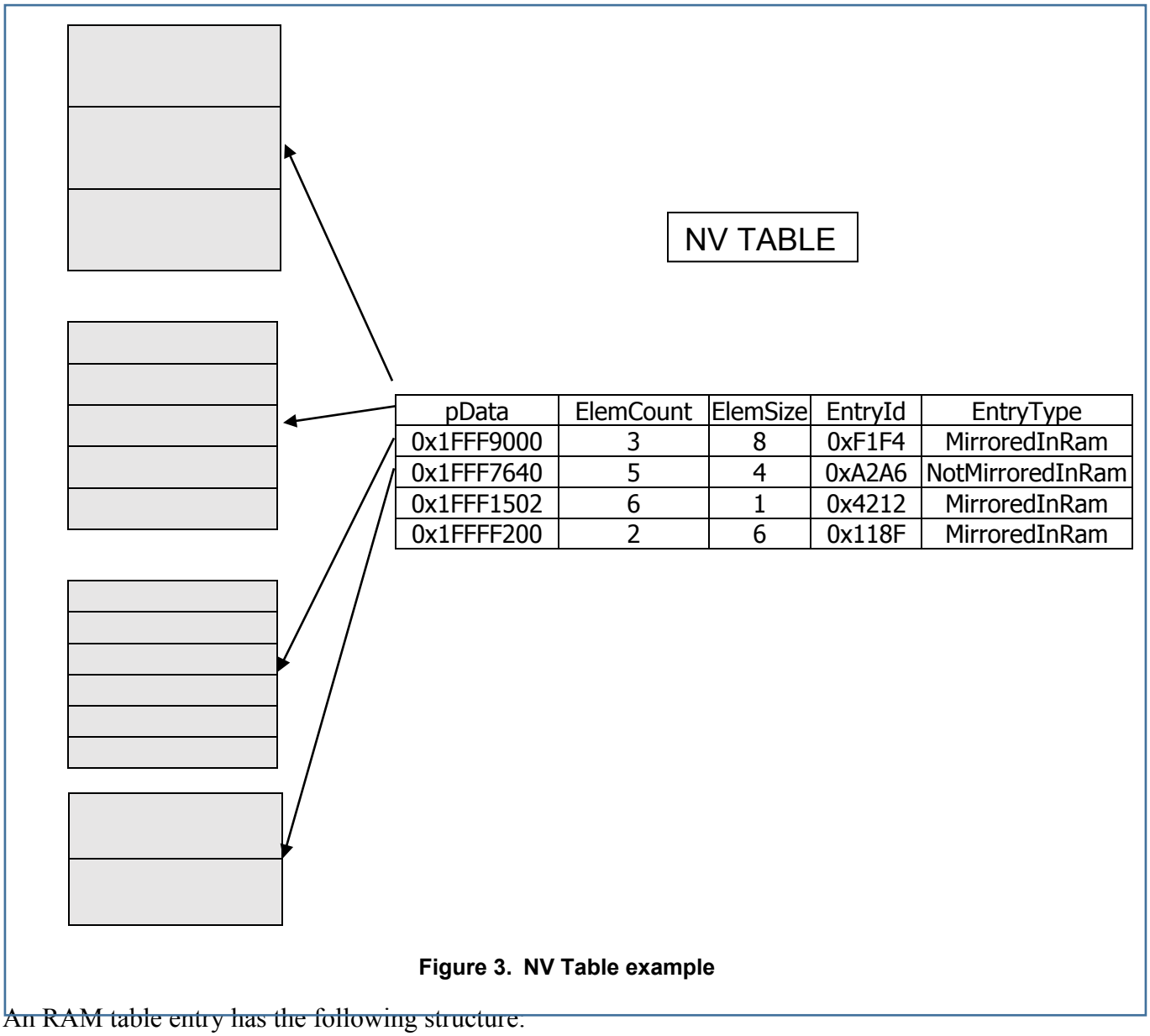


Figure 3. NV Table example

An RAM table entry has the following structure:

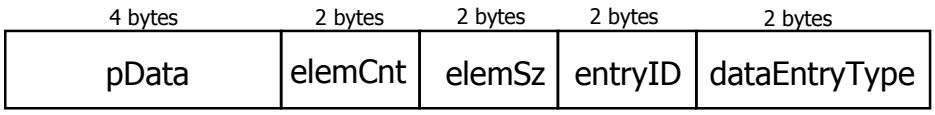


Figure 4. The structure of a NV table entry

Where:

- **pData** is a pointer to the 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
- **dataEntryType** is a 16-bit value representing the type of entry (mirrored/unmirrored)

For mirrored datasets, pData must point directly to the RAM data, for unmirrored it must be a double pointer to a vector of pointers, each pointer in this table pointing to a RAM/FLASH area.

Mirrored datasets require the data to be kept in RAM all the time, while unmirrored datasets will have the dataset entries either in flash or in ram at a time. Unmirrored entries must be enabled by the application by setting gUnmirroredFeatureSet_d to 1.

The last entry in the RAM table must have the entryID set to gNvEndOfTableId_c.

Mirrored data entry example:

pData	0x1FFF8000
elemCnt	4
elemSz	10
entryID	1
dataEntryType	gNVM_MirroredInRam_c

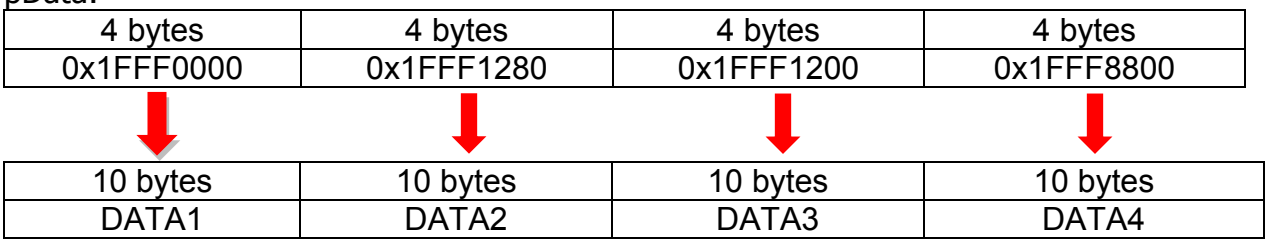
pData:

10 bytes	10 bytes	10 bytes	10 bytes
DATA1	DATA2	DATA3	DATA4

Unmirrored data entry example:

pData	0x1FFF8000
elemCnt	4
elemSz	10
entryID	1
dataEntryType	gNVM_NotMirroredInRam_c

pData:



When the data pointed to by the table entry pointer (pData) have changed (entirely or just a single element), the upper layers can call the appropriate API function that will make a request to the storage system to save the modified data. All the save operations (except for the synchronous save and atomic save), as well as the page erase and page copy operations, are performed on system idle task. The application must create a task that calls NvIdle in an infinite loop. It should be created with OSA_PRIORITY_IDLE, but the application may choose another priority.

The saves are done in one virtual page – the active page. After a save is performed on an unmirrored dataset, pData will point to a FLASH location and the RAM pointer will be freed. Because of this behavoiur, the effective data should always be allocated using the memory management module.

The active page contains information about the records that it holds, as well as the records. The storage system can save individual elements of a table entry or the entire table entry if the table entry. Unmirrored datasets can only have individual saves.

On mirrored datasets, the save/restore functions must receive the pointer to the RAM data. For example, if the application must save the third element in the above vector, it should send 0x1FFF8000 + 2 * elemSz.

For unmirrored datasets, the application must send the pointer that points to the area where the data is located. For example, if the application must save the third element in the above vector, it should send 0x1FFF8000 + 2 * sizeof(void*).

The page validity is guaranteed by the page counter. The page counter is a 32-bit value, and it 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 when the system is formatted, or every time a page is full and a new record cannot be written into that page. Before being erased, the full page is first copied (only the most recent saves), and erased afterwards.

The validity of the Meta Information Tag (MIT) (and therefore of a record) is guaranteed by the MIT start and stop validation bytes. These two bytes must be equal to consider the record referred by the MIT valid. Furthermore, the value of these bytes indicates the type of the record: single element or entire table entry.

The nonvolatile storage system allows dynamic changes of the table within the RAM memory, as follows:

- remove table entry
- register table entry

A new table entry can be successfully registered if there is at least one entry previously removed, or if the NV table contains uninitialised table entries, declared explicitly to register new table entries at runtime. A new table entry can also replace an existing one if register table entry is called with overwrite set to true. This functionality is disabled by default and must be enabled by the application by setting gNvUseExtendedFeatureSet_d to 1.

The layout of an active page is depicted below:

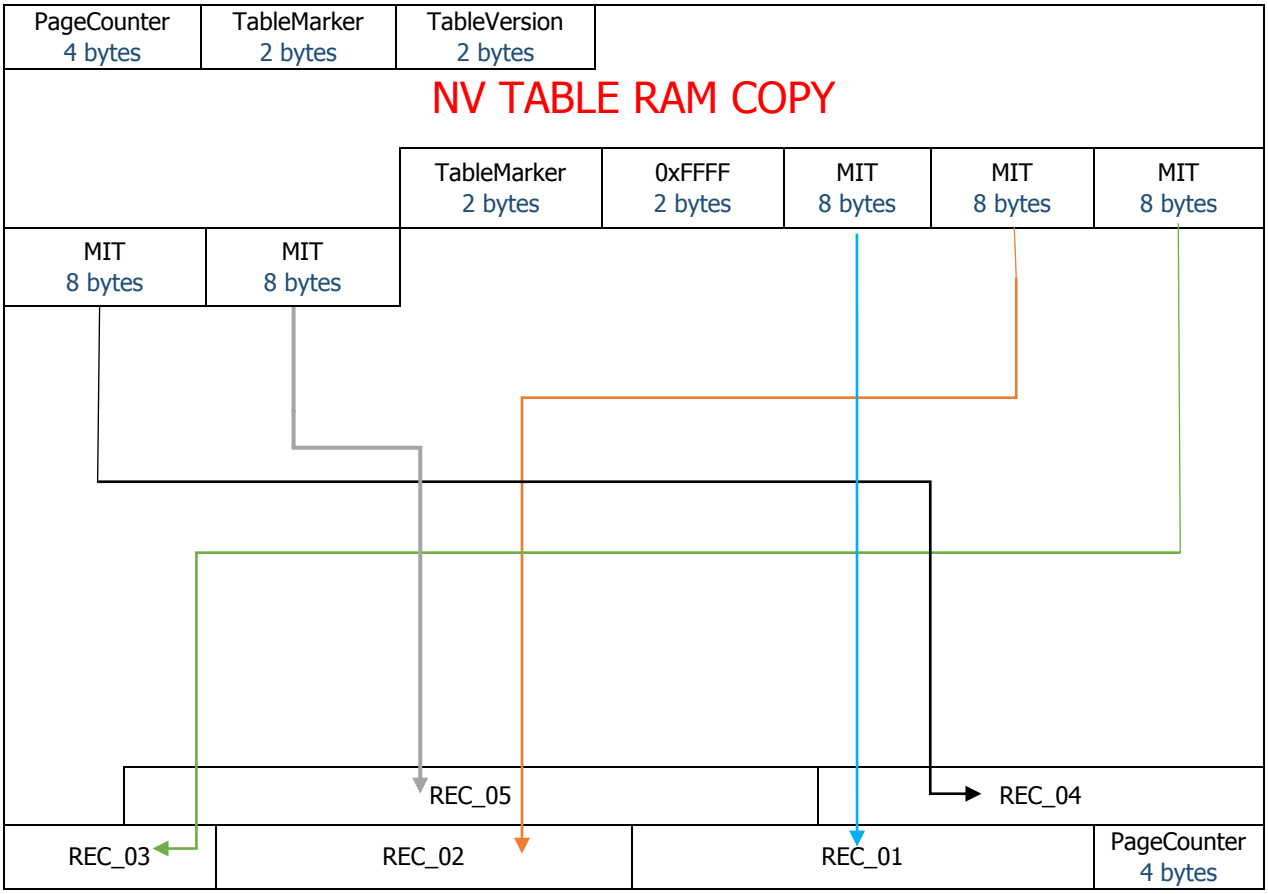


Figure 5. The layout of a flash active page

As shown above, the table stored in the RAM memory is copied into the flash active page, just after the table version. The “table start” and “table end” are marked by the table markers. The data pointers from the RAM are not copied. A flash copy of a RAM table entry has the following structure

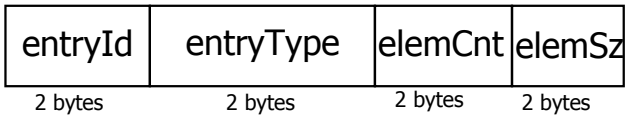


Figure 6. The structure of the flash copy of a RAM table entry

Where:

- **entryID** is the ID of the table entry
- **entryType** represents the type of the entry (mirrored/unmirrored)
- **elemCnt** is the elements count of that entry
- **elemSz** is the size of a single element

This copy of the RAM table in flash is used to determine, whether the RAM table has changed.

The table marker has a value of 0x4254 (“TB” if read as ASCII codes), and marks the beginning and end of the NV table copy.

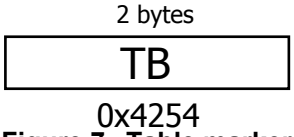


Figure 7. Table marker

After the end of the RAM table copy, the Meta Information Tags (MITs) follow. Each MIT is used to store information related to one record. An MIT has the following structure:

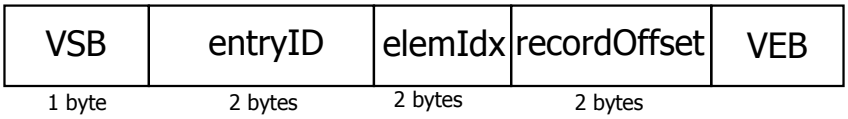


Figure 8. Meta Information Tag

Where:

- **VSB** is the validation start byte
- **entryID** 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 equal to VEB. If the MIT refers to a single-element record type, then **VSB=VEB=0xAA**. If the MIT refers to a full table entry record type (all elements from a table entry), then **VSB=VEB=0x55**.

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

In the following example, there are five datasets (one color for each dataset); to be a good 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**, **R6** and **R11** are ‘single’ record types.
- **R2** – full record type; **R15** – single record type
- **R5**, R**13** – full record type; **R10**, R**12** – single record type
- **R8** – full record type
- **R7**, **R9**, **R14**, **R16** – full record type

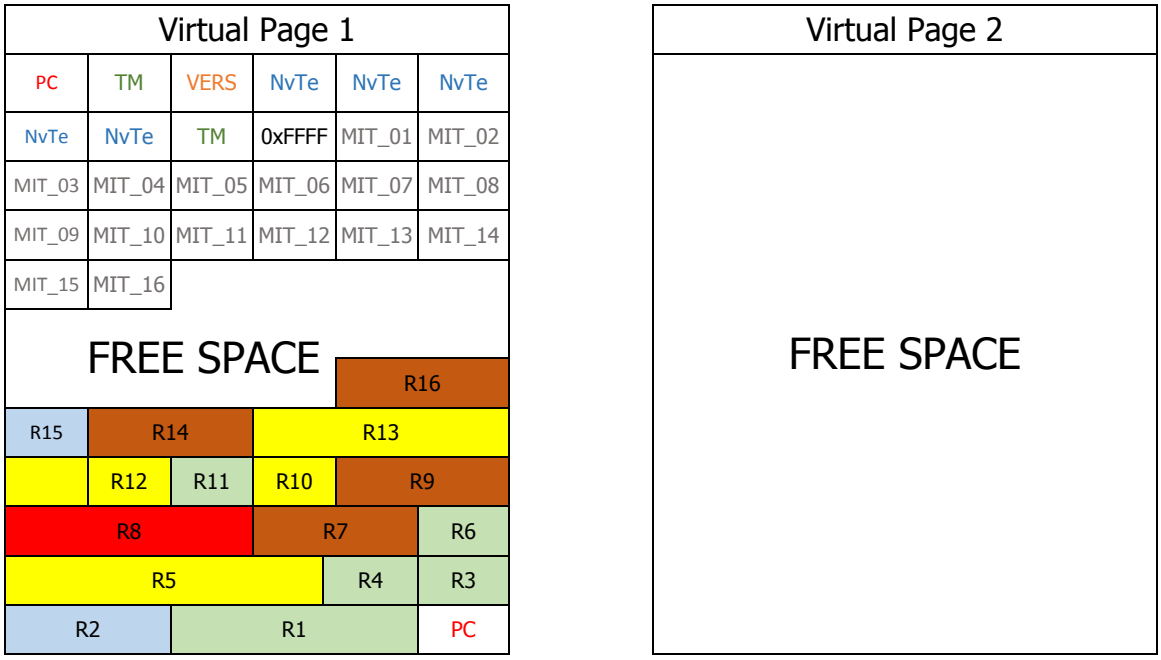


Figure 9. Virtual page 1 free space is not large enough for saving a new dataset

As shown above, the **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 takes 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 five ‘full’ record types, one for each dataset.

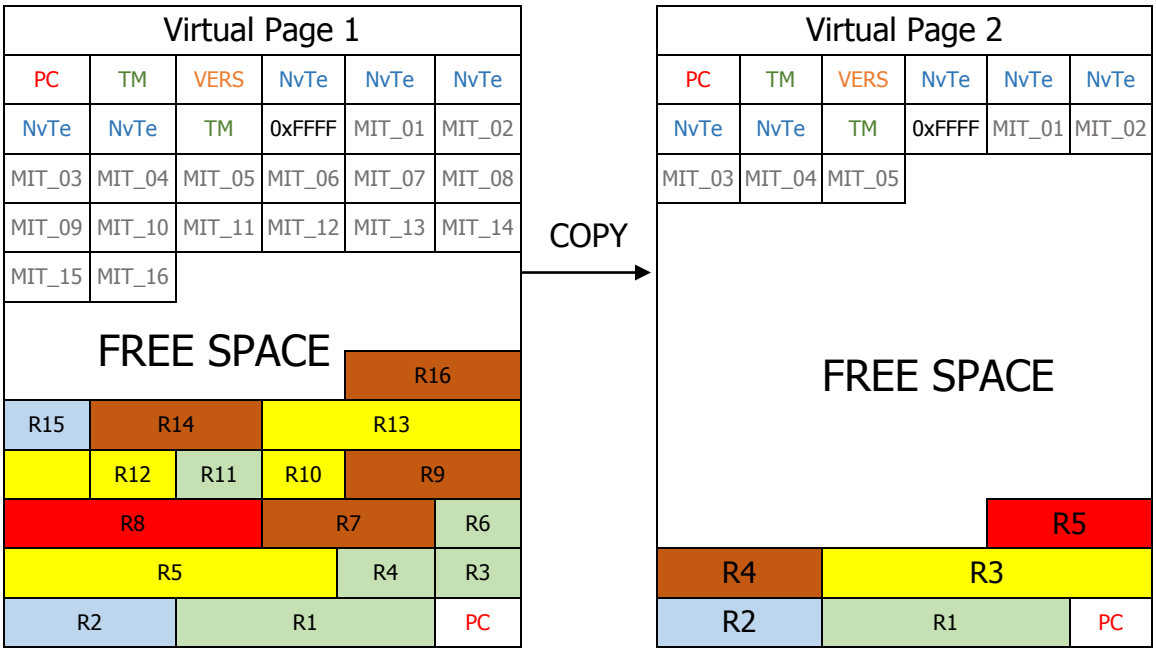


Figure 10. Page copy

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).

If there are any difference between the RAM and FLASH tables, the application must call RecoverNvEntry for each entry that is different from its RAM copy to recover the entry data (Id, Type, ElemSz, ElemCnt) from flash before calling NvInit. The application is responsible for allocating pData and changing the RAM entry. It can choose to ignore the flash entry, if the entry is not desired. If any entry from RAM differes from its flash equivalent at init, a page copy will be triggered that will ignore the entries that are different (the data stored in those entries will be lost).

The application can check if the RAM table was updated (i.e. the MCU program was changed and the RAM table was updated) using the function GetFlashTableVersion and compare the result with the constant gNvFlashTableVersion_c. If the versions are different, NvInit will detect the update and automaticly upgrade the flash table. The upgrade process will trigger a page copy that will move the flash data from the active page to other one. It will keep the entries that were not modified intact and it will move the entries that had their elements count changed as following:

-if the RAM element count is smaller than the FLASH element count, then the upgrade will only copy as much elements are in RAM

-if the RAM element count is larger than the FLASH element count, then the upgrade will copy all the data from FLASH and it will fill the remaining space with data from RAM

If the entry size is changed, the entry will not be copied. Any entryIds that are present in FLASH and not present in RAM will also not be copied.

This functionality is not supported if gNvUseExtendedFeatureSet_d is not set to 1.

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. The FlexNVM can be partitioned to support a simulated EEPROM subsystem. 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 created and stored in the FlexNVM. The EEPROM endurance capability can exceed 10 000 000 cycles.

The NonVolatile Memory platform component supports this feature, and implements the associated routines. From the user point of view, the API is exactly the same. All you have to do is to enable the FlexNVM support, by the means 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 into the FlexNVM memory. The wear-leveling algorithm is implemented and controlled by the hardware, and it is not under user control. The Meta Information Tags store information about the ID of the NV table entry and the offset to the record itself, related to FlexRAM start address. After all datasets (RAM table entries) are written to FlexRAM, and when an element of a given dataset (or the entire dataset) is changed and the upper layer decides that the change must be saved, the corresponding record is overwritten (partially or totally). In the example below, there are eight NV table entries (datasets) saved in FlexRAM. The '**meta08**' refers to the '**rec08**' record, and using the information stored by the '**meta08'**, the entire record '**rec08'** can be read / updated, or just a single element ('**e5'** in the below example).

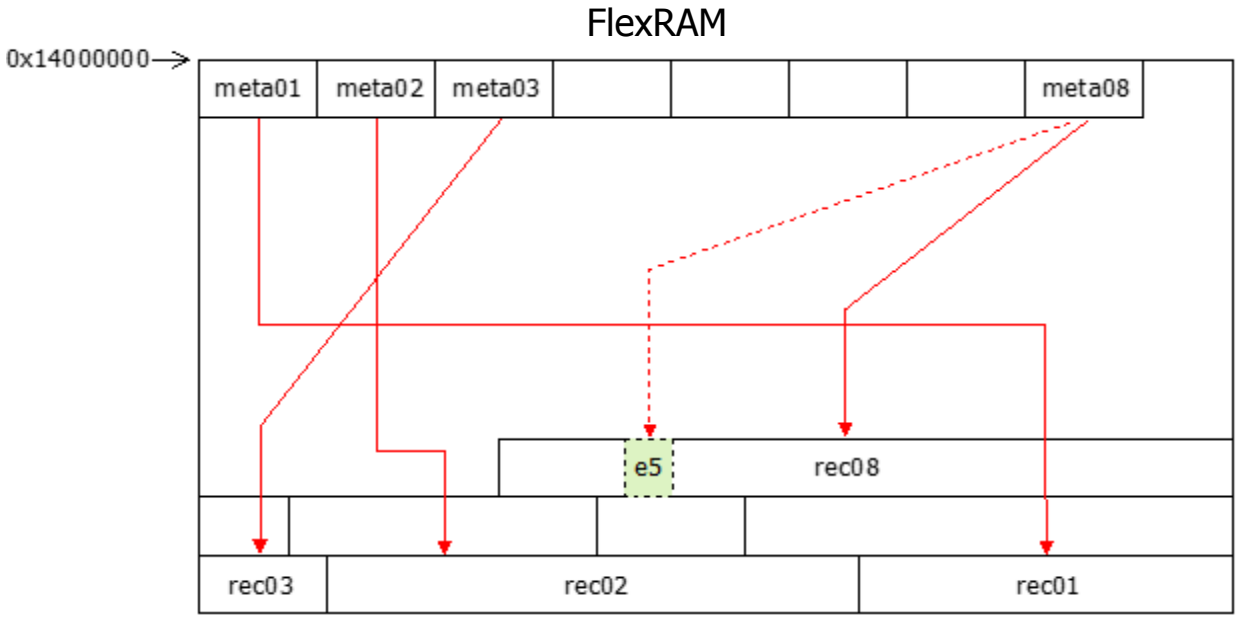


Figure 11. FlexRAM memory used to store NV table entries

If Flex NVM is enabled, unmirrored datasets are not be supported.

3.5.4. Application Programming Interface

3.5.4.1. Constant macro definitions

Name:

`gNvStorageIncluded_d`

Description:

If set to TRUE, it enables the whole functionality of the nonvolatile storage system. By default, it is set to FALSE (no code or data is generated for this module).

Name:

`gNvUseFlexNVM_d`

Description:

If set to TRUE, it enables the FlexNVM functionality of the nonvolatile storage system. By default, it is set to FALSE. If FlexNVM is used, the standard nonvolatile storage system is disabled.

Name:

`gNvFragmentation_Enabled_d`

Description:

Macro used to enable / disable the fragmented saves / restores (a particular element from a table entry can be saved or restored). It is set to FALSE by default.

Name:

`gNvUseExtendedFeatureSet_d`

Description:

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

- Remove existing NV table entries
- Register new NV table entries
- Table upgrade

It is set to FALSE by default.

Name:

`gUnmirroredFeatureSet_d`

Description:

Macro used to enable unmirrored datasets. It is set to 0 by default.

Name:

`gNvTableEntriesCountMax_c`

Description:

This constant defines the maximum count of the table entries (datasets) that the application is going to use. It is set to 32 by default.

Name:

`gNvRecordsCopiedBufferSize_c`

Description:

This constant defines the size of the buffer used by the page copy function, when the copy operation performs defragmentation. The chosen value must be a power of 2 and bigger than the maximum number of elements stored in any of the table entries. It is set by default to 64.

Name:

`gNvCacheBufferSize_c`

Description:

This constant defines the size of the cache buffer used by the page copy function, when the copy operation does not perform defragmentation. The chosen value must be a multiple of 4. It is set by default to 64.

Name:

`gNvMinimumTicksBetweenSaves_c`

Description:

This constant defines the minimum timer ticks between dataset saves (in seconds). It is set to 4 by default.

Name:

`gNvCountsBetweenSaves_c`

Description:

This constant defines the number of calls to ‘NvSaveOnCount’ between dataset saves. It is set to 256 by default.

Name:

`gNvInvalidDataEntry_c`

Description:

Macro used to mark a table entry as invalid in the NV table. Default value is 0xFFFFU.

Name:

`gNvFormatRetryCount_c`

Description:

Macro used to define the maximum retries count value for the format operation. It is set to 3 by default.

Name:

`gNvPendigSavesQueueSize_c`

Description:

Macro used to define the size of the pending saves queue. It is set to 32 by default.

Name:

`gFifoOverwriteEnabled_c`

Description:

Macro used to allow overwriting of older entries in the pending saves queue, if it is full. It is set to FALSE by default.

Name:

`gNvMinimumFreeBytesCountStart_c`

Description:

Macro used to define the minimum free space at init. If the free space is smaller than this value, a page copy will be triggered. It is set by default to 128.

Name:

`gNvEndOfTableId_c`

Description:

Macro used to define the ID of the end-of-table entry. It is set to 0xFFFEU by default. No valid entry should use this ID.

Name:

```
gNvTableMarker_c
```

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. It is set to 0x4254U by default.

Name:

```
gNvFlashTableVersion_c
```

Description:

Macro used to define the flash table version. It is used to determine if the RAM table was updated. It is set to 1 by default. The application should modify this every time the RAM table is updated and the data from NVM is still required.

3.5.5. User-defined data type definitions

Name:

```
typedef uint16_t NvSaveInterval_t;
```

Description:

Data type definition used by dataset save on interval function.

Name:

```
typedef uint16_t NvSaveCounter_t;
```

Description:

Data type definition used by dataset save on count function.

Name:

```
typedef uint16_t NvTableEntryId_t;
```

Description:

Data type definition for a 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.

3.5.5.1. Flash Management API primitives

NvModuleInit

Prototype:

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

Description:

This function is used to initialize the Flash Management module. It indirectly initializes the Flash HAL driver and gets the active page ID. It initializes internal state variables and counters. If the RAM entries are different from FLASH entries, a page copy will be triggered and the different entries will be skipped in the process. It also handles ram table changes (for example the MCU program was changed and the RAM table was updated) by automaticly doing a table upgrade. To trigger this behavior, the application must change gNvFlashTableVersion_c.

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
```

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 to by the ptrData argument, as soon as the NVM_Task() is the highest priority ready-to-run task in the system. If ptrData belongs to an unmirrored dataset, after the save the RAM pointer will be freed and pData will point to the FLASH backup. No other saves can be made while the data is in flash.

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, whether the entire table entry is saved, or just the element pointed to by ptrData

Returns:

One of the following:

`gNVM_OK_c`

`gNVM_ModuleNotInitialized_c`

`gNVM_NullPointer_c`

`gNVM_InvalidTableEntry_c`

`gNVM_SaveRequestRejected_c`

NvSaveOnInterval

Prototype:

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

Description:

This function saves the specified dataset no more often than at a given time interval. If it has been at least that long since the last save, this function causes a save as soon as the NVM_Task() is the highest-priority ready-to-run task in the system. If ptrData belongs to an unmirrored dataset, after the save the RAM pointer will be freed and pData will point to the FLASH backup. No other saves can be made while the data is in flash. If ptrData belongs to a mirrored dataset, a full save will be made, otherwise only the element indicated by ptrData will be saved. If the function is called before a previous save on interval was processed, the call will be ignored.

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
```

NvSaveOnCount

Prototype:

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

Description:

This function decrements a counter that is associated with the dataset specified by the function argument, and when that counter reaches zero, the dataset is saved as soon as the NVM_Task() is the highest-priority ready-to-run task in the system. If ptrData belongs to an unmirrored dataset, after the save the RAM pointer will be freed and pData will point to the FLASH backup. No other saves can be made while the data is in flash. If ptrData belongs to a mirrored dataset, a full save will be made, otherwise only the element indicated by ptrData will be saved.

Parameters:

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

Returns:

None.

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 is completed.

Parameters:

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

Returns:

None.

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 is completed.

Parameters:

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

Returns:

None.

NvSyncSave

Prototype:

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

Description:

This 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. If ptrData belongs to an unmirrored dataset, after the save, the RAM pointer will be freed and pData will point to the FLASH backup. Also, saveAll is ignored and only individual elements can be saved. No other saves can be made while the data is in flash.

Parameters:

Name	Type	Direction	Description
ptrData	void*	IN	Pointer to the table entry to be saved
saveAll	bool_t	IN	Specifies whether the entire table entry shall be saved, or just the pointed element
ignoreCriticalSectionFlag	bool_t	IN	If set to TRUE, the function ignores the critical section flag, and performs 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`

NvAtomicSave

Prototype:

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

Description:

This 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 ignores the critical section flag, and performs 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
```

NvTimerTick

Prototype:

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

Description:

This function processes NvSaveOnInterval() requests. If the call of this function counts 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 datasets tick counters have not yet counted down to zero, or FALSE if all data set tick counters have reached zero. This function is called automaticly inside the module to process interval saves, but it can be called from the application if required.

Parameters:

Name	Type	Direction	Description
countTick	bool_t	IN	See API description

Returns:

See description.

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 the flash memory, it will be restored to RAM NV Table.

For unmirrored datasets, the function will only restore a pointer to the flash location of the entry. In order to restore the data from flash to ram, the application will have to call NvMoveToRam.

Parameters:

Name	Type	Direction	Description
ptrData	void*	IN	A pointer to a NV table entry / element to be restored with the data from flash.
restoreAll	bool_t	IN	A flag used to indicate if the entire table entry shall be restored, or just a single element (indicated by ptrData). If ptrData points to an unmirrored dataset, the flag is set to false internally.

Returns:

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

NvSetCriticalSection

Prototype:

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

Description:

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

Parameters:

None.

Returns:

None.

NvClearCriticalSection

Prototype:

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

Description:

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

Parameters:

None.

Returns:

None.

NvIdle

Prototype:

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

Description:

This function processes the NvSaveOnIdle() and NvSaveOnCount() requests. It also checks if the internal timer made a tick and determines if any save on interval should be processed. It also does page copy and erase. It must be called by the NVM task.

Parameters:

None.

Returns:

None.

NvIsDataSetDirty

Prototype:

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

Description:

This function checks if the table entry specified by the function argument is dirty(a save is pending on the specified dataset).

Parameters:

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

Returns:

TRUE if the specified table entry is dirty / FALSE otherwise.

NvGetPageStatistics

Prototype:

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

Description:

Retrieves the virtual pages statistics (how many times each virtual page has been erased). The function is not available on FlexNVM.

Parameters:

Name	Type	Direction	Description
ptrStat	NVM_Statistics_t*	OUT	Pointer to a memory location where the statistics are to be stored

Returns:

None.

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
```

NvRegisterTableEntry

Prototype:

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

Description:

This function enables you 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 (unused / previously erased table entry). If overwrite is TRUE, the old table entry with the specified ID will be overwritten. This will trigger a page copy. Extended functionality must be enabled if this function is required.

Parameters:

Name	Type	Direction	Description
ptrData	void*	IN	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 of the table entry to be registered / updated
elemSize	uint16_t	IN	The size of one element of the table entry
dataEntryType	uint16_t	IN	The type of the entry to be registered
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. Otherwise, if overwrite is set to FALSE, the data will be placed in the first free position in the table.

Returns:

One of the following:

`gNVM_OK_c`

`gNVM_ModuleNotInitialized_c`

`gNVM_AlreadyRegistered`

`gNVM_RegisterFailure_c`

NvEraseEntryFromStorage

Prototype:

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

Description:

This function removes the table entry specified by the function argument, ptrData. A page copy will be triggered and the data associated with the entry will not be copied. It sets entrySize and entryCount to 0 and pData to NULL. EntryId remains unchanged. For unmirrored datasets, it is recommended to use NvErase, because that function can delete individual elements. Extended functionality must be enabled if this function is required.

Parameters:

Name	Type	Direction	Description
ptrData	void*	IN	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`

GetFlashTableVersion

Prototype:

```
uint16_t GetFlashTableVersion
(
  void
);
```

Description:

This function returns the version stored in the flash table or 0 if no table is detected. Extended functionality must be enabled if this function is required.

Parameters:

Name	Type	Direction	Description
–	–	–	–

Returns:

See description.

RecoverNvEntry

Prototype:

```
NVM_Status_t RecoverNvEntry
(
    uint16_t index,
    NVM_DataEntry_tag *entry
);
```

Description:

This function reads the flash entry data for a specified dataset. It is used to determine before init if some flash entries are different from the RAM counterparts. If the application desires to use the data stored in the flash table, it should copy the members returned by this function in their respective RAM entry and allocate pData. Extended functionality must be enabled if this function is required.

Parameters:

Name	Type	Direction	Description
index	uint16_t	IN	The index of the RAM entry that needs to be restored
entry	NVM_DataEntry_tag*	OUT	The entry data from flash

Returns:

One of the following:

`gNVM_OK_c`

`gNVM_RestoreFailure_c`

NvMoveToRam

Prototype:

```
NVM_Status_t NvMoveToram
(
    void** ppData
);
```

Description:

This function moves the data pointed to by ppData from flash to RAM (allocates space and copies the data). It can only move a single element. It changes pData in the ram table to point to the new location. If the specified element is already in RAM, the function will cancel any pending saves and return. Unmirrored functionality must be enabled if this function is required.

Parameters:

Name	Type	Direction	Description
ppData	void**	IN	Pointer to the table entry to be moved

Returns:

One of the following:

`gNVM_OK_c`

`gNVM_PointerOutOfRange_c`

`gNVM_IsMirroredDataSet_c`

`gNVM_InvalidTableEntry_c`

`gNVM_NoMemory_c`

`gNVM_Error_c`

NvErase

Prototype:

```
NVM_Status_t NvErase
(
    void** ppData
);
```

Description:

This erases an unmirrored dataset from flash. It does not trigger page copy, but writes a new record in the NVM memory that specifies that this element was removed. It sets pData for the specified element to NULL. It can only erase a single element. If the specified element is in RAM, it will be freed but no changes will be made in flash. Unmirrored functionality must be enabled if this function is required.

Parameters:

Name	Type	Direction	Description
ppData	void**	IN	Pointer to the table entry to be erased

Returns:

One of the following:

`gNVM_OK_c`

`gNVM_PointerOutOfRange_c`

`gNVM_IsMirroredDataSet_c`

`gNVM_InvalidTableEntry_c`

3.5.6. Production Data Storage

3.5.6.1. Overview

Different platforms/boards need board/network node specific settings settings (i.e. IEEE addresses, radio calibration values specific to the node) to function according to design. For this purpose, the last FLASH sector is reserved, and contains hardware specific parameters for production data storage. These parameters pertain to the network node as a distinct entity, e.g. silicon monted on a PCB in a specific configuration, rather than to just the silicon itself.

This sector is reserved from linker file, through the *FREESCALE_PROD_DATA* section and it should be read/written only thorough the API detailed below.

NOTE

This sector is not erased/written at code download time and it is not updated via over-the-air firmware update procedures, in order to preserve the respective node-specific data, regardless of the firmware running on it.

Usually, when the *hardware_init()* function is called, the hardware parameters are loaded from the flash sector into the *gHardwareParameters* RAM structure. The default behavior is that if the respective sector or addresses within the respective sector are empty in the flash, the firmware will not take into account the respective values corresponding to node-specific data.

3.5.6.2. Constant Definitions

Name:

```
extern uint32_t FREESCALE_PROD_DATA_BASE_ADDR[];
```

Description:

This symbol is defined in the linker file, and it specify the start address of the *FREESCALE_PROD_DATA* section.

3.5.6.3. Data type definitions

Name:

```
typedef PACKED_STRUCT hardwareParameters_tag{
    uint8_t  reserved[50];
    uint8_t  ieee_802_15_4_address[8];
    uint8_t  bluetooth_address[6];
    uint32_t xtalTrim;
    uint32_t gInternalStorageAddr;
}hardwareParameters_t;
```

Description:

Defines the structure of the HW dependent information.

NOTE

Some members of this structure may be ignored on a specific board/silicon configuration. Also, new members may be added for implementation specific purposes, but the backwards compatibility must be maintained.

3.5.6.4. API Primitives

NV_ReadHWParameters

Prototype:

```
uint32_t NV_ReadHWParameters
(
    hardwareParameters_t *pHwParams
);
```

Description:

Load the HW dependent information into a RAM data structure.

Parameters:

Name	Type	Direction	Description
pHwParams	hardwareParameters _t*	[OUT]	Pointer to a RAM location where the information will be stored

Returns:

The error code.

NV_WriteHWParameters

Prototype:

```
uint32_t NV_WriteHWParameters
(
    hardwareParameters_t *pHwParams
);
```

Description:

Store the HW dependent information into Flash. It is recommended to use a read-modify-write sequence when HW specific data needs to be updated.

Parameters:

Name	Type	Direction	Description
pHwParams	hardwareParameters _t*	[IN]	Pointer to a data structure containing HW dependent information to be written to Flash

Returns:

The error code.

3.6. Random number generator

3.6.1. Overview

The RNG module is part of the framework, and it 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 hardware acceleration is present, the RNG module uses a software algorithm. The initial seed for this algorithm represents the device unique ID (SIM_UID registers) by default. You 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 hardware support options for the RNG.

Name:

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

Description:

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

Name:

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

Description:

Defines the status codes for the 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

RNG_Init ()

Prototype:

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

Description:
Initializes the hardware RNG module.

Parameters:

None.

Returns:

Status of the RNG module.

RNG_GetRandomNo ()

Prototype:

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

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

Parameters:

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

Returns:

None.

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.

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. Must 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 the relevant system data in flash. Then an external tool will 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

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 the ‘DEBUG’ instruction, which makes the Debugger stop and display the current program counter.

Parameters:

Name	Type	Direction	Description
id	panicId_t	[IN]	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]	Provides 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

ResetMCU()

Prototype:

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

Description:
Resets the MCU.

Parameters:

None.

Returns:

None.

3.9. Serial manager

3.9.1. Overview

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

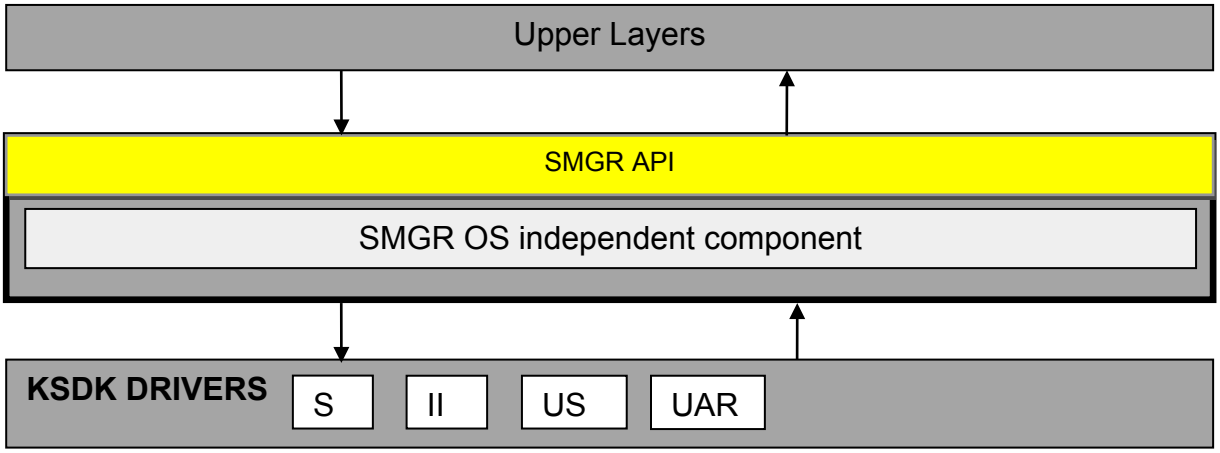


Figure 12. Serial manager overview

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 you 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 is blocked until the TX operation is completed. Note that a synchronous TX is also buffered, and it completes and unblocks 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 must 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 does not block, but an error message which will be handled by the user is returned instead.

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 / disables input parameter checking.

Name:

```
#define gSerialMgr_BlockSenderOnQueueFull_c 1
```

Description:

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

Name:

```
#define gSerialMgrIICAddress_c 0x76
```

Description:

Defines the address to be used for I^2C.

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 you want to print a hex number, you 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 whether the TX is blocking or not.

Name:

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

Description:

Defines whether 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

SerialManager_Init ()

Prototype:

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

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

Parameters:

None.

Returns:

None.

Serial_InitInterface ()

Prototype:

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

Description:
Initializes 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 is reached
- gSerial_InternalError_c if an internal error occurred
- gSerial_Success_c if the operation was successful

Serial_SetBaudRate ()

Prototype:

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

Description:
Sets the communication speed of 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

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

Serial_SetRxCallBack ()

Prototype:

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

Description:
Sets a pointer to a function that is called when data are 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

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

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

Serial_SyncWrite ()

Prototype:

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

Description:

Transmits a data buffer synchronously. The task blocks 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

Serial_AsyncWrite ()

Prototype:

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

Description:

Transmits 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

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 waits 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

Serial_PrintHex ()

Prototype:

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

Description:

Prints a number in hexadecimal format to the serial interface. The task waits 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 bytes of 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

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

Serial_EnableLowPowerWakeup ()

Prototype:

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

Description:
Configures the enabled hardware modules of the given interface type as a wake-up source from the 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

Serial_DisableLowPowerWakeup ()

Prototype:

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

Description:
Configures the enabled hardware modules of the given interface type as modules without wake-up 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

Serial_IsWakeUpSource ()

Prototype:

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

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

Parameters:

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

Returns:

TRUE if a module of the given interface type was the wake-up source, FALSE otherwise.

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 both 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 FSCI Black Boxes at various layers. In this setup, you can run numerous commands to test the Black Box application services and interfaces.

The FSCI enables common service features for each device, and enables 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 opcodes to specific interfaces. After doing so, any packet coming from that interface with the same opcode will trigger a callback execution. Two or more entities cannot register the same opcode on the same interface, but they can do so on different interfaces. For example, two MAC instances can register the same opcodes, one over UARTA, and the other over UARTB. This way you can use Test Tool to communicate with 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 it is designed to offer the best communication reliability. The Black Box device is expecting messages in little-endian format, and it responds with messages in little-endian format.

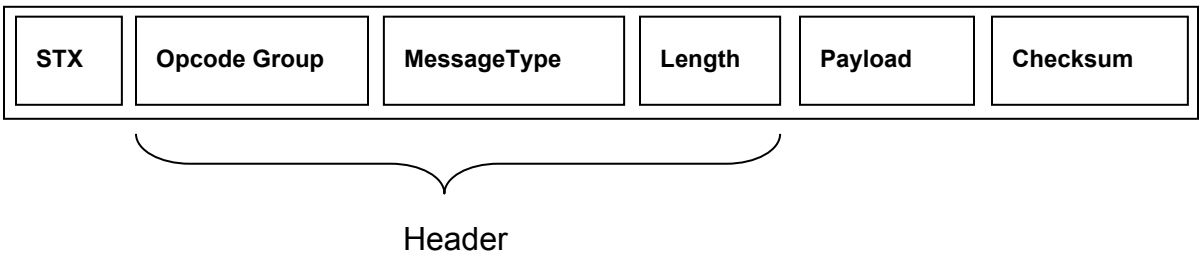


Figure 13. Packet structure

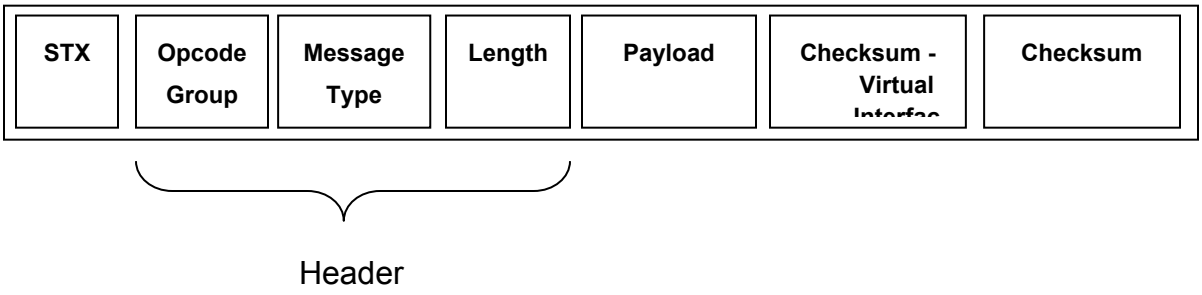


Figure 14. Packet structure when virtual interfaces are used

Table 2. Packet field description

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 (for example 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 must 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.

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:

Defines the message types that the 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

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 an initialization structure

Returns:

None.

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 handles the received message
param	void*	[IN]	Pointer to a parameter that is provided inside the OG Handler function
fsciInterface	uint32_t	[IN]	The interface ID on which the callback must 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

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 (for example, a status message)
fsciInterface	uint32_t	[IN]	The interface on which the data must be printed

Returns:

Returns the status of the call process.

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 is accessed

Returns:

None.

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.

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]	Specifies whether the data should be transmitted asap
pSrc	void*	[IN]	Pointer to the data location
len	index_t	[IN]	Size of the data

Returns:

None.

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.

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 must be printed

Returns:

Returns the status of the call process.

FSCI_transmitFormatedPacket

Prototype:

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

Description:

Send a packet over the serial interface, after computing the 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 must be sent

Returns:

None.

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.

FSCI_Error

Prototype:

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

Description:

Sends a 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 must be sent

Returns:

None.

3.10.6. FSCI Host

FSCI Host is a functionality that allows separation at a certain stack layer between two entities, usually two boards running separate layers of a stack.

Support is provided for functionality at the MAC layer, for example, MAC / PHY layers of a stack are running as black box on a board, and MAC higher layers running on another. The higher layers send and receive serial commands to and from the MAC black box using the FSCI set of operation codes and groups.

The protocol of communication between the two is the same, and the current level of support is provided for:

- FSCI_MsgResetCPUReqFunc – sends a CPU reset request to black box
- FSCI_MsgWriteExtendedAdrReqFunc – configures MAC extended address to black box
- FSCI_MsgReadExtendedAdrReqFunc – N/A

The approach on the Host interfacing a black box using synchronous primitives is by default the polling of the FSCI_receivePacket function, until the response is received from the black box. The calling task will poll whenever the task is being scheduled. This is required because a stack synchronous primitive requires that the response of that request is available in the context of the caller right after the SAP call has been executed.

The other option besides polling, available for RTOS environments, is to use an event mechanism. The calling task blocks waiting for the event that is sent from the Serial Manager task when the response is available from black box. This option is disabled by default. The disadvantage of this option is that the primitive cannot be received from another black box through a serial interface, as the task being blocked would be the Serial Manager task, and reaches a deadlock as it cannot be released again.

3.10.7. FSCI usage example

Example 4. 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 );
```

Example 5. 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 );
```

Example 6. 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 be four bytes aligned.

Both implementations are supplied in a 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 hardware support options for SecLib.

Name:

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

Description:

Defines the default hardware 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

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.

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.

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 to

Returns:

None.

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 to

Returns:

None.

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 to

Returns:

None.

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 to

Returns:

None.

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 to

Returns:

None.

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 to

Returns:

None.

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 to
pTag	uint8_t*	[OUT]	Pointer to the location to store the 128-bit tag to

Returns:

Operation status.

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 to
pTag	uint8_t*	[OUT]	Pointer to the location to store the 128-bit tag to

Returns:

Operation status.

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
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 to
pCbcMac	uint8_t*	[IN/OUT]	Encryption: pointer to the location to store the authentication code to Decription: pointer to the location of the received authentication code
flags	uint8_t	[IN]	Bit0 – 0 encrypt / 1 decrypt

Returns:

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

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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 input 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

This 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:

Lists 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

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.

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.

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.

ListAddHead

Prototype:

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

Description:
Inserts an element to 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.

ListRemoveHead

Prototype:

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

Description:
Unlinks an 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.

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.

ListGetNext

Prototype:

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

Description:

Gets the next element in the 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.

ListGetPrev

Prototype:

```
listElementHandle_t ListGetPrev
(
    listElementHandle_t element
);
```

Description:

Gets the previous element in the list.

Parameters:

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

Returns:

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

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.

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 the 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.

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.

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

Example 7. 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

This 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

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:
Copies 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.

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 into 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.

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:
Compares 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, or FALSE otherwise.

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:

Resets 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.

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 they are 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.

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 into an unaligned 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.

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.

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 the content of buffers is equal, or FALSE otherwise.

FLib_GetMax

Prototype:

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

Description:
Returns the maximum values 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.

FLib_GetMin

Prototype:

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

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

Parameters:

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

Returns:

The minimum values of arguments a and b.

3.14. Low-power library

The low-power module (LPM) simplifies the process of putting a Kinetis-based wireless network node into the low-power or sleep modes to preserve battery life. In IEEE® 802.15.4 terminology, only End-Devices (EDs) can sleep while the Coordinators must remain awake to route packets on behalf of other nodes. In Bluetooth® LE terminology, the peripheral sensor nodes must always go to sleep, while the central nodes can remain awake, depending on the application.

3.14.1. Kinetis MKW2xD and MCR20A Transceiver Low-power Library

The low-power module (LPM) offers access to interface functions and macros that allow the developer to perform the following actions:

- Enable or disable the device to enter low-power modes.
- Check whether the device is enabled to enter low-power modes.
- Put the device into one of the low-power modes.
- Configure which low-power modes the device must use.
- Set the low-power state duration in milliseconds or symbols.
- Configure the wake-up sources from the low-power modes.
- Power off the device.
- Reset the device.
- Get the system reset status.

The files `PWR.c`, `PWRLib.h`, `PWRLib.c`, `PWR_Configuration.h`, `PWR_Interface.h`, and `LPMConfig.h` comprise the low-power library. The LPM API provides access to three types of low-power modes: Sleep, Deep Sleep, and Power Off. The Sleep / Deep Sleep modes are statically configured at compile-time through properties located in `PWR_Configuration.h` and `LPMConfig.h`. The configuration for the Sleep and Deep Sleep functions corresponds to a combination of processor-radio's low-power modes. The API functions determine whether the low-power configuration is valid, and activate the selected low-power mode. Wake up sources are also statically configured at compile time. The duration of the low-power state can be configured for some of the low-power modes (in milliseconds or symbols) statically at the compile-time or at the run-time. After waking up, the LPM functions return the wake-up reason (if applicable).

The low-power library must be initialized by the application, by calling the *PWR_CheckForAndEnterNewPowerState_Init()* function. The code for entering the low-power state must be placed in the Idle task or another low-priority application task. The application can also choose to stay awake by using the *PWR_DisallowDeviceToSleep()* function.

NOTE

With the exception of *PWR_DisallowDeviceToSleep()* and *PWR_AllowDeviceToSleep()*, do not call the low-power functions directly in the application. The idle task already contains the proper code for entering deep or light sleep as appropriate, in a way that coordinates with the entire system.

Example 8. Entering low-power state from a low-priority task

```
...

if( PWR_CheckIfDeviceCanGoToSleep() )
{
    PWR_EnterLowPower();
}

...
```

3.14.2. Low-power library properties

The following list is not an exhaustive list of properties, but it includes the most important ones. For the entire list, see `PWR_Configuration.h`.

cPWR_UsePowerDownMode

Set *cPWR_UsePowerDownMode* to TRUE to enable the whole low-power code. The whole low-power code and variables are compiled out (the low-power library API functions still exist) if *cPWR_UsePowerDownMode* is FALSE. Low power can be disabled at run-time using *PWR_DisallowDeviceToSleep()* from the application, however, this will not save code space.

NOTE

If the *gMPibRxOnWhenIdle_c* MAC PIB is set to TRUE, the device never enters the low-power mode.

cPWR_DeepSleepMode

Set cPWR_DeepSleepMode to the appropriate deep-sleeping mode for the application.

Deep sleep modes for Kinetis MKW2xD and MCR20A transceiver-based platforms

- Mode 1
  - MCU / Radio low-power modes:
    - MCU sleep mode is the VLLS2 mode. Only portion of SRAM_U remains powered on. Wake-up goes through the Reset sequence.
    - Radio sleep mode is hibernate mode.
  - Wake-up sources:
    - User push-button switches interrupt using low-leakage wake-up unit (LLWU) module.
- Mode 2
  - MCU / Radio low-power modes:
    - MCU sleep mode is VLLS2 mode. Only portion of SRAM_U remains powered on. Wake-up goes through the Reset sequence.
    - Radio in hibernate mode.
  - Wake-up sources:
    - Low-power timer (LPTMR) interrupt using low-leakage wake-up unit (LLWU) module.

    - Low-power timer (LPTMR) wake-up timeout: fixed at compile time.
    - Low-power timer (LPTMR) clock source: internal low-power oscillator (LPO).
    - Low-power timer (LPTMR) available resolutions: 1 ms, 2 ms, 4 ms, 8 ms, 16 ms, 32 ms, 64 ms, 128 ms, 256 ms, 512 ms, 1024 ms, 2048 ms, 4096 ms, 8192 ms, 16384 ms, 32768 ms, 65536 ms. See `PWRLib.h` for details.
- Mode 3
  - MCU / Radio low-power modes:
    - MCU sleep mode is VLLS2 mode. Only portion of SRAM_U remains powered on. Wake-up goes through the Reset sequence.
    - Radio in hibernate mode.
  - Wake-up sources:
    - Low-power timer (LPTMR) interrupt using low-leakage wake-up unit (LLWU) module.
    - Low-power timer (LPTMR) wake-up timeout: fixed at compile time.
    - Low-power timer (LPTMR) clock source: ERCLK32K (secondary external reference clock; 32.768 kHz crystal is connected to the RTC oscillator).
    - Low-power timer (LPTMR) available resolutions: 125 / 4096 ms, 125 / 2048 ms, 125 / 1024 ms, 125 / 512 ms, 125 / 256 ms, 125 / 128 ms, 125 / 64 ms, 125 / 32 ms, 125 / 16 ms, 125 / 8 ms, 125 / 4 ms, 125 / 2 ms, 125 ms, 250 ms, 500 ms, 1000 ms, 2000 ms. See `PWRLib.h` for details.
- Mode 4
  - MCU / Radio low-power modes:
    - MCU sleep mode is VLLS2 mode. Only portion of SRAM_U remains powered on. Wake-up goes through the Reset sequence.
    - Radio in hibernate mode.
  - Wake-up sources:
    - Real-time clock (RTC) interrupt using low-leakage wake-up unit (LLWU) module.
    - Real-time clock (RTC) wake-up timeout – fixed at compile time.
    - Real-time clock (RTC) clock source – 32.768 kHz crystal connected to real-time clock (RTC) oscillator.
    - Real-time clock (RTC) available resolutions: 1 s.
- Mode 5
  - MCU / Radio low-power modes:
    - MCU sleep mode is VLLS2 mode. Only portion of SRAM_U remains powered on. Wake-up goes through the Reset sequence.
    - Radio in hibernate mode.
  - Wake-up sources:
    - User push-button switches interrupt using low-leakage wake-up unit (LLWU) module.
    - Low-power timer (LPTMR) interrupt using low-leakage wake-up unit (LLWU) module.

  - Low-power timer (LPTMR) wake-up timeout – fixed at compile time.
  - Low-power timer (LPTMR) clock source – internal low-power oscillator (LPO).
  - Low-power timer (LPTMR) possible resolutions: 1 ms, 2 ms, 4 ms, 8 ms, 16 ms, 32 ms, 64 ms, 128 ms, 256 ms, 512 ms, 1024 ms, 2048 ms, 4096 ms, 8192 ms, 16384 ms, 32768 ms, 65536 ms. See `PWRLib.h` for details.

- Mode 6
  - MCU / Radio low-power modes:
    - MCU sleep mode is VLLS2 mode. Only portion of SRAM_U remains powered on. Wake-up goes through the Reset sequence.
    - Radio in hibernate mode.
  - Wake-up sources:
    - User push-button switches interrupt using low-leakage wake-up unit (LLWU) module.
    - Low-power timer (LPTMR) interrupt using low-leakage wake-up unit (LLWU) module.
    - Low-power timer (LPTMR) wake-up timeout: fixed at compile time.
    - Low-power timer (LPTMR) clock source: ERCLK32K (secondary external reference clock;
    - 32.768 kHz crystal connected to RTC oscillator).
    - Low-power timer (LPTMR) possible resolutions: 125 / 4096 ms, 125 / 2048 ms, 125 / 1024 ms, 125 / 512 ms, 125 / 256 ms, 125 / 128 ms, 125 / 64 ms, 125 / 32 ms, 125 / 16 ms, 125 / 8 ms, 125 / 4 ms, 125 / 2 ms, 125 ms, 250 ms, 500 ms, 1000 ms, 2000 ms. See `PWRLib.h` for details.
- Mode 7
  - MCU / Radio low-power modes:
    - MCU sleep mode is VLLS2 mode. Only portion of SRAM_U remains powered on. Wake-up goes through the Reset sequence.
    - Radio in hibernate mode.
  - Wake-up sources:
    - User push-button switches interrupt using low-leakage wake-up unit (LLWU) module.
    - Real-time clock (RTC) interrupt using low-leakage wake-up unit (LLWU) module.
    - Real-time clock (RTC) wake-up timeout – fixed at compile time.
    - Real-time clock (RTC) clock source – 32.768 kHz crystal connected to the real-time clock (RTC) oscillator.
    - Real-time clock (RTC) possible resolutions: 1 s.
- Mode 8
  - MCU / Radio low-power modes:
    - MCU sleep mode is VLLS2 mode. Only portion of SRAM_U remains powered on. Wake-up goes through the Reset sequence.
    - Radio in hibernate mode.

  - Wake-up sources:
    - User push-button switches interrupt using low-leakage wake-up unit (LLWU) module.
    - Low-power timer (LPTMR) interrupt using low-leakage wake-up unit (LLWU) module.
    - Low-power timer (LPTMR) Wake-up timeout: configurable at run-time.
    - Low-power timer (LPTMR) clock source: internal low-power oscillator (LPO).
    - Low-power timer (LPTMR) possible resolutions: fixed to 2 ms (125 802.15.4 PHY symbols).
- Mode 9
  - MCU / Radio low-power modes:
    - MCU sleep mode is low-leakage stop (LLS) mode.
    - Radio in hibernate mode.
  - Wake-up sources:
    - User push-button switches interrupt using low-leakage wake-up unit (LLWU) module.
    - Low-power timer (LPTMR) interrupt using low-leakage wake-up unit (LLWU) module.
    - Low-power timer (LPTMR) wake-up timeout: fixed at compile time.
    - Low-power timer (LPTMR) clock source: internal low-power oscillator (LPO).
    - Low-power timer (LPTMR) available resolutions: 1 ms, 2 ms, 4 ms, 8 ms, 16 ms, 32 ms, 64 ms, 128 ms, 256 ms, 512 ms, 1024 ms, 2048 ms, 4096 ms, 8192 ms, 16384 ms, 32768 ms, 65536 ms. See `PWRLib.h` for details.
- Mode 10
  - MCU / Radio low-power modes:
    - MCU sleep mode is low-leakage stop (LLS) mode.
    - Radio in hibernate mode.
  - Wake-up sources:
    - User push-button switches interrupt using low-leakage wake-up unit (LLWU) module.
    - Low-power timer (LPTMR) interrupt using low-leakage wake-up unit (LLWU) module.
    - Low-power timer (LPTMR) wake-up timeout: fixed at compile time.
    - Low-power timer (LPTMR) clock source: ERCLK32K (secondary external reference clock; 32.768 kHz crystal connected to RTC oscillator).
    - Low-power timer (LPTMR) possible resolutions: 125 / 4096 ms, 125 / 2048 ms, 125 / 1024 ms, 125 / 512 ms, 125 / 256 ms, 125 / 128 ms, 125 / 64 ms, 125 / 32 ms, 125 / 16 ms, 125 / 8 ms, 125 / 4 ms, 125 / 2 ms, 125 ms, 250 ms, 500 ms, 1000 ms, 2000 ms. See `PWRLib.h` for details.
- Mode 11
  - MCU / Radio low-power modes:
    - MCU sleep mode is low-leakage stop (LLS) mode.
    - Radio in hibernate mode.

  - Wake-up sources:
    - User push-button switches interrupt using low-leakage wake-up unit (LLWU) module.
    - Real-time clock (RTC) interrupt using low-leakage wake-up unit (LLWU) module.
    - Real-time clock (RTC) wake-up timeout: fixed at compile time.
    - Real-time clock (RTC) clock source: 32.768 kHz crystal connected to real-time clock (RTC) oscillator.
    - Real-time clock (RTC) possible resolutions: 1 s.
- Mode 12
  - MCU / Radio low-power modes:
    - MCU sleep mode is low-leakage stop (LLS) mode.
    - Radio in hibernate mode.
  - Wake-up sources:
    - User push-button switches interrupt using low-leakage wake-up unit (LLWU) module.
    - Low-power timer (LPTMR) interrupt using low-leakage wake-up unit (LLWU) module.
    - Low-power timer (LPTMR) wake-up timeout: configurable at run-time.
    - Low-power timer (LPTMR) clock source: internal low-power oscillator (LPO).
    - Low-power timer (LPTMR) possible resolutions: fixed to 2 ms (125 802.15.4 PHY symbols).
- Mode 13
  - MCU / Radio low-power modes:
    - MCU sleep mode is very-low-power stop (VLPS) mode.
    - Radio in doze mode.
  - Wake-up sources:
    - UART module interrupt.
    - User push-button switches interrupt.
    - Radio timer interrupt.
    - Radio timer Wake-up timeout: configurable at run-time.
    - Radio timer possible resolutions: fixed to 16 μs (one 802.15.4 PHY symbol).
    - Set *gSerialMgrUseUart_c* to TRUE if UART module is used as wake-up source.
    - Set *gKeyBoardSupported_d* to TRUE if user push-button switches are used as wake-up source. Only the switches that are connected to the LLWU will wake up the MCU from the LLS modes.
- Mode 14
  - MCU / Radio low-power modes:
    - MCU in LLS mode. Wake-up goes through the Reset sequence.
    - Radio in hibernate mode.

  - Wake-up sources:
    - User push-button switches interrupt using low-leakage wake-up unit (LLWU) module.
    - Low-power timer (LPTMR) interrupt using low-leakage wake-up unit (LLWU) module.
    - LPTMR wake-up timeout: the shortest timeout period of any active LPTMR-based timer (see `Timer.c`)
    - LPTMR resolution is fixed to 1 ms
- Mode 15
  - MCU / Radio low-power modes:
    - MCU in LLS mode. Wake-up goes through the Reset sequence.
    - Radio power mode is managed by MAC.
  - Wakeup sources:
    - User push-button switches interrupt using low-leakage wake-up unit (LLWU) module.
    - Low-power timer (LPTMR) interrupt using low-leakage wake-up unit (LLWU) module.
    - LPTMR wake-up timeout: the shortest timeout period of any active LPTMR-based timer (see `Timer.c`)
    - LPTMR resolution is fixed to 1 ms

cPWR_SleepMode

Leave the cPWR_SleepMode as TRUE (1). This option saves considerable power when the system is running. It works by entering an MCU halt when the system enters the idle task, and it wakes instantly when an interrupt occurs (UART, keyboard, timer expires, and so on). This can save 30 % of power at run-time. The radio enters the auto-doze mode, and the MCU is in WAIT mode. The clock on the radio is still running, but the transceiver is disabled.

cPWR_LPTMRClockSource

The cPWR_LPTMRClockSource represents the low-power timer (LPTMR) clock source.
The cPWR_LPTMRClockSource can be set to one of the following values:

- cLPTMR_Source_Int_LPO_1KHz – internal 1 KHz low-power oscillator (LPO)
- cLPTMR_Source_Ext_ERCLK32K – ERCLK32K (secondary external reference clock; 32.768 kHz crystal connected to the RTC oscillator).

cPWR_LPTMRTickTime

The cPWR_LPTMRTickTime represents the low-power timer (LPTMR) resolution, and depends on the LPTMR clock source (cPWR_LPTMRClockSource).

When the cPWR_LPTMRClockSource is set to cLPTMR_Source_Int_LPO_1KHz, the cPWR_LPTMRTickTime can be set to one of the following values:

- cLPTMR_PRS_00001ms – the LPTMR resolution is set to 1 ms
- cLPTMR_PRS_00002ms – the LPTMR resolution is set to 2 ms

- cLPTMR_PRS_00004ms – the LPTMR resolution is set to 4 ms
- cLPTMR_PRS_00008ms – the LPTMR resolution is set to 8 ms
- cLPTMR_PRS_00016ms – the LPTMR resolution is set to 16 ms
- cLPTMR_PRS_00032ms – the LPTMR resolution is set to 32 ms
- cLPTMR_PRS_00064ms – the LPTMR resolution is set to 64 ms
- cLPTMR_PRS_00128ms – the LPTMR resolution is set to 128 ms
- cLPTMR_PRS_00256ms – the LPTMR resolution is set to 256 ms
- cLPTMR_PRS_00512ms – the LPTMR resolution is set to 512 ms
- cLPTMR_PRS_01024ms – the LPTMR resolution is set to 1024 ms
- cLPTMR_PRS_02048ms – the LPTMR resolution is set to 2048 ms
- cLPTMR_PRS_04096ms – the LPTMR resolution is set to 4096 ms
- cLPTMR_PRS_08192ms – the LPTMR resolution is set to 8192 ms
- cLPTMR_PRS_16384ms – the LPTMR resolution is set to 16384 ms
- cLPTMR_PRS_32768ms – the LPTMR resolution is set to 32768 ms
- cLPTMR_PRS_65536ms – the LPTMR resolution is set to 65536 ms

When the cPWR_LPTMRClockSource is set to cLPTMR_Source_Ext_ERCLK32K, the cPWR_LPTMRTickTime can be set to one of the following values:

- cLPTMR_PRS_125_div_by_4096ms – the LPTMR resolution is set to 125 / 4096 ms
- cLPTMR_PRS_125_div_by_2048ms – the LPTMR resolution is set to 125 / 2048 ms
- cLPTMR_PRS_125_div_by_1024ms – the LPTMR resolution is set to 125 / 1024 ms
- cLPTMR_PRS_125_div_by_512ms – the LPTMR resolution is set to 125 / 512 ms
- cLPTMR_PRS_125_div_by_256ms – the LPTMR resolution is set to 125 / 256 ms
- cLPTMR_PRS_125_div_by_128ms – the LPTMR resolution is set to 125 / 128 ms
- cLPTMR_PRS_125_div_by_64ms – the LPTMR resolution is set to 125 / 64 ms
- cLPTMR_PRS_125_div_by_32ms – the LPTMR resolution is set to 125 / 32 ms
- cLPTMR_PRS_125_div_by_16ms – the LPTMR resolution is set to 125 / 16 ms
- cLPTMR_PRS_125_div_by_8ms – the LPTMR resolution is set to125 / 8 ms
- cLPTMR_PRS_125_div_by_4ms – the LPTMR resolution is set to 125 / 4 ms
- cLPTMR_PRS_125_div_by_2ms – the LPTMR resolution is set to 125 / 2 ms
- cLPTMR_PRS_0125ms – the LPTMR resolution is set to 125 ms
- cLPTMR_PRS_0250ms – the LPTMR resolution is set to 250 ms
- cLPTMR_PRS_0500ms – the LPTMR resolution is set to 500 ms
- cLPTMR_PRS_1000ms – the LPTMR resolution is set to 1000 ms
- cLPTMR_PRS_2000ms – the LPTMR resolution is set to 2000 ms

cPWR_TMRTicks

The cPWR_TMRTicks represents the deep sleep duration or LPTMR / RTC timer wake-up timeout.

When the LPTMR (see cPWR_DeepSleepMode) is used, the time to deep sleep is cPWR_TMRTicks multiplied by cPWR_LPTMRTickTime.

When the RTC (see cPWR_DeepSleepMode) is used, the time to deep sleep is cPWR_TMRTicks multiplied by RTC timer resolution, which is one second.

cPWR_DeepSleepDurationMs

The default deep sleep duration in milliseconds used by modes 8, 12, and 13. For these modes, the duration of sleep can be changed at runtime using *PWR_SetDeepSleepTimeInMs()* or *PWR_SetDeepSleepTimeInSymbols()*.

cPWR_CallWakeupStackProcAfterDeepSleep

Enablement of external call to a procedure each time that DeepSleep is exited. The application is responsible for implementing the *DeepSleepWakeupStackProc()* function:

```
extern void DeepSleepWakeupStackProc(void);
```

gPWR_EnsureOscStabilized_d

This define configures, whether to test if the RTC oscillator has started on entering low power, for the low-power modes that use it.

cPWR_LVD_Enable

The use of low-voltage detection has the following possibilities:

- 0: Don’t use low-voltage detection at all
- 1: Use polled => Check is made each time the function is called.
- 2: A minutes software timer used for handling when to poll the LVD, according to the cPWR_LVD_Ticks constant
- 3: LVDRE are set to hold the MCU in reset while the VLVDL condition is detected

cPWR_LVD_Ticks

How often to check the LVD level when cPWR_LVD_Enable == 2. This is the number of minutes before the voltage is checked (consumes current and time).

3.14.3. Low-power library API

PWR_CheckForAndEnterNewPowerState_Init

Prototype

```
void PWR_CheckForAndEnterNewPowerState_Init( void );
```

Description

Initializes the low-power module.

PWR_CheckIfDeviceCanGoToSleep

Prototype

```
bool_t PWR_CheckIfDeviceCanGoToSleep( void );
```

Description

Checks the flag, and ensures whether it is allowed to go to low power at this time. Always ensure that this returns TRUE before calling PWR_EnterLowPower().

PWR_EnterLowPower

Prototype

```
PWRLib_WakeupReason_t PWR_EnterLowPower(void);
```

Description

This is the function called by the application to set the device into the low-power mode. It decides which power state is activated, based on the current state of the system.

PWR_DisallowDeviceToSleep

Prototype

```
void PWR_DisallowDeviceToSleep (void);
```

Description

Sets a critical section to prevent the system from entering low power. Use this if the device must stay awake, for example, during ZigBee commissioning or during a sensor reading.

PWR_AllowDeviceToSleep

Prototype

```
void PWR_AllowDeviceToSleep (void);
```

Description

Clears the critical section set by PWR_DisallowDeviceToSleep(). This is a “counting semaphore.” There should be one call to PWR_AllowDeviceToSleep() for every call to PWR_DisallowDeviceToSleep().

3.14.4. Kinetis Wireless Dual Mode (Bluetooth® LE and IEEE® 802.15.4) Microcontrollers Low-power Library Overview

3.14.4.1. Overview

The dual mode Kinetis wireless microcontrollers (e.g. MKW40Z) have a specific architecture of a system-on-chip. This architecture contains various elements of data link layer hardware acceleration for the wireless protocols (Bluetooth® LE and IEEE® 802.15.4) and a DC to DC converter. These features of the SoC mandate a certain functionality of the low power managing firmware modules.

3.14.4.2. When/How to Enter Low Power

The system should enter low power when the entire system is idle and all software layers agree on that. An idle task which must have the lowest priority in the system should be defined and used to enter low power. So, the system enters low power on idle task, which runs only when there are no events for other tasks. The user must place a call to the function PWR_EnterLowPower in this idle task through a construction like bellow:

Example 9.

```
if (PWR_CheckIfDeviceCanGoToSleep())
{
    wakeupReason = PWR_EnterLowPower();
}
```

Each software layer/entity running on the system can prevent it from entering low power by calling PWR_DisallowDeviceToSleep(). The system will stay awake until all software layers that called PWR_DisallowDeviceToSleep() call back PWR_AllowDeviceToSleep()and the system reaches idle task. The MCU will enter either sleep or deep sleep depending on the type of the timers started. Low power timers are the only timers that do not prevent the system from entering deep sleep. If any other timers are started the MCU will enter in sleep instead of deep sleep. So, if you want the MCU to enter deep sleep, stop all timers other than the low power ones. Be aware that functions like LED_StartFlash starts timers which will prevent the system from entering deep sleep.

3.14.4.3. Deep sleep modes

Four low power modes have been developed so far and the user can switch between them at run time using PWR_ChangeDeepSleepMode function. The default low power mode is selected by the cPWR_DeepSleepMode define value in LPMConfig.h header file.

3.14.4.4. Deep sleep mode 1

This low power mode was designed to be used when the BLE stack is active.

In this mode, the MCU enters LLS3 and BLE Link Layer enters deep sleep. The SoC wakes up from this mode by SW2 (PTA18), SW3 (PTA19), or by BLE Link Layer wakeup interrupt (BLE_LL reference clock reaches wake up instance register) using LLWU module. LPTMR timer is used to measure the time MCU spends in deep sleep in order to synchronize low power timers at wakeup. There are two ways to use this mode:

1. The BLE stack decides it can enter low power and calls PWR_AllowDeviceToSleep(). If no other software entity prevents the system from entering deep sleep (all software layers that called PWR_DisallowDeviceToSleep() have called back PWR_AllowDeviceToSleep()) and the system reaches idle task, PWR_EnterLowPower function is entered and the system prepares for entering low power mode 1. BLE Link layer status is checked and found not to be in deep sleep. A function from BLE stack is called to get the nearest instant at which the BLE Link layer needs to be running again and the wakeup instant register in the BLE Link layer is programmed with this value. The BLE link layer is then put in deep sleep and the MCU enters LLS3.
2. The BLE stack decides it can enter low power and calls PWR_BLE_EnterDSM(wakeupInstant) followed by PWR_AllowDeviceToSleep(). In this way the BLE Link layer is put to deep sleep immediately, the MCU remaining to enter LLS3 on idle task. If no other software entity prevents the system from entering deep sleep (all software layers that called PWR_DisallowDeviceToSleep() have called back PWR_AllowDeviceToSleep()), and the system reaches idle task, PWR_EnterLowPower function is entered and the system prepares to complete entering low power mode 1. BLE Link layer status is checked and found to be in deep sleep, so the MCU puts itself in LLS3 and deep sleep mode 1 finally reached.

With a timeout calculated as cPWR_BLE_LL_OscStartupDelay + cPWR_BLE_LL_OffsetToWakeupInstant before BLE link layer reference clock register reaches the value in wakeup register, BLE Link Layer wakes up the entire SoC and the system resumes its activity. The two defines above can be found in PWR_Configuration.h header file.

3.14.4.5. Deep sleep mode 2

This low power mode was designed to be used when the BLE stack is idle. In this mode, the MCU enters LLS3 and BLE Link Layer enters deep sleep. The SoC wakes up from this mode by SW2 (PTA18), SW3 (PTA19), or by BLE Link Layer wakeup interrupt (BLE_LL reference clock register reaches wake up instance register) using LLWU module. LPTMR timer is used to measure the time MCU spends in deep sleep in order to synchronize low power timers at wakeup. The deep sleep duration can be configured at compile time using cPWR_DeepSleepDurationMs define in PWR_Configuration.h header file or at run time calling PWR_SetDeepSleepTimeInMs(deepSleepTimeTimeMs) function. The main drawback of this mode is that the maximum deep sleep duration is limited to 40959 ms.

3.14.4.6. Deep sleep mode 3

This mode was developed to overcome mode 2 maximum deep sleep duration limitation. It is intended to be used when the BLE stack is idle also.

In this mode, the MCU enters LLS3 and BLE Link Layer remains idle. The SoC wakes up from this mode by SW2 (PTA18), SW3 (PTA19), or by LPTMR timeout using LLWU module. LPTMR timer is also used to measure the time MCU spends in deep sleep in order to synchronize low power timers at wakeup. The deep sleep duration can be configured at compile time using cPWR_DeepSleepDurationMs define in PWR_Configuration.h header file or at run time calling PWR_SetDeepSleepTimeInMs(deepSleepTimeTimeMs) function. The maximum configurable deep sleep duration in this mode is 65535000 ms (18.2h).

3.14.4.7. Deep sleep mode 4

This is the lowest power mode of all. It was designed to be used when the BLE stack is idle.

In this mode, the MCU enters VLLS0/VLLS1 and BLE Link Layer remains idle. The SoC wakes up from this mode by SW2 (PTA18) and SW3 (PTA19) using LLWU module. No synchronization for low power timers is made since this deep sleep mode is exited through reset sequence. There are two defines that configures this mode:

cPWR_DCDC_InBypass configures the VLLS mode used. If this define is TRUE the MCU enters VLLS0, otherwise MCU enters VLLS1 since VLLS0 is not allowed in DCDC buck or boost mode.

cPWR_POR_DisabledInVLLS0. This define only has meaning if cPWR_DCDC_InBypass is TRUE so the MCU enters VLLS0 mode. If TRUE, this define disables POR circuit in VLLS0 making this deep sleep mode lowest power mode possible.

3.14.4.8. Deep sleep mode 5

This is the lowest power mode which allows RAM retention. It was designed for those applications which requires RAM retention over low power.

In this mode, the MCU enters VLLS2 and BLE Link Layer remains idle. 4k of RAM, from the address 0x20000000 to the address 0x20000FFF are retained. The SoC wakes up from this mode by SW2 (PTA18) , SW3 (PTA19) and DCDC power switch (when DCDC is in buck mode) using LLWU module. No synchronization for low power timers is made since this deep sleep mode is exited through reset sequence.

3.14.4.9. Deep sleep mode 6

This low power mode was developed to save some power while the radio is on. Its most common use case is with the radio in Rx waiting for a packet. Upon receiving the packet the radio wakes up the MCU.

In this mode, the MCU enters STOP mode and the radio (BLE or 802.15.4) maintains its state. Any module capable to produce an interrupt can wake up the MCU: SW2 (PTA18), SW3 (PTA19), DCDC power switch (when DCDC is in buck mode), LPTMR timeout, Radio Interrupt (LL or 802.15.4),

UART, etc. LPTMR timer is also used to measure the time MCU spends in deep sleep in order to synchronize low power timers at wakeup. The deep sleep timeout can be configured at compile time using cPWR_DeepSleepDurationMs define in PWR_Configuration.h header file or at run time calling PWR_SetDeepSleepTimeInMs(deepSleepTimeTimeMs) function. The maximum configurable deep sleep timeout in this mode is 65535000 ms (18.2h).

3.14.4.10. Low Power API

Configuration Macros

Name

```
#define cPWR_UsePowerDownMode                   TRUE
```

Description

Enables/Disables low power related code and variables.

Location

PWR_Configuration.h

Name

```
#define cPWR_DeepSleepMode                      4
```

Description

Configures default deep sleep mode.

Location

PWR_Configuration.h

Name

```
#define cPWR_DeepSleepDurationMs                30000
```

Description

Configures default deep sleep duration. It only has meaning for deep sleep modes 2 and 3.

Location

PWR_Configuration.h

Name

```
#define cPWR_BLE_LL_Enable              TRUE
```

Description

Enables/Disables low power code related with BLE Link Layer.

Location

PWR_Configuration.h

Name

```
#define cPWR_Zigbee_Enable              FALSE
```

Description

Enables/Disables low power code related with Zigbee stack.

Location

PWR_Configuration.h

Name

```
#define cPWR_BLE_LL_OffsetToWakeupInstant                   (8)
```

Description

Number of slots (625us) when BLE Link Layer hardware needs to exit from deep sleep mode before BLE reference clock register reaches wake up instant register.

Location

PWR_Configurations.h

Name

```
#define cPWR_BLE_LL_OscStartupDelay                         (8)
```

Description

Period that BLE Link Layer must wait after a system clock request to be sure the oscillator is running and stable. This is in X.Y format where X is in terms of number of BT slots (625 us) and Y is in terms of number of clock periods of 16 KHz clock input, required for RF oscillator to stabilize the clock output after oscillator is turned ON. In this period the clock is assumed to be unstable, and so the controller does not turn on the clock to internal logic until this period is over. This means, the wake up from deep sleep mode must account for this delay before the wakeup instant.

Osc_startup_delay[7:5] is number of slots(625us)

Osc_startup_delay[4:0] is number of clock periods of 16KHz clock

(Warning: Max. value of Osc_startup_delay [4:0] supported is 9. Therefore do not program value greater than 9)

Location

PWR_Configurations.h

Name

```
#define cPWR_DCDC_InBypass                                  (1)
```

Description

Set this define 1 if DCDC is in bypass mode or 0 otherwise. It will configure deep sleep mode 4 to put the MCU in VLLS0 (when 1) or VLLS1 (when 0) since VLLS0 is only supported in bypass mode.

Location

PWR_Configurations.h

Name

```
#define cPWR_POR_DisabledInVLLS0                            (1)
```

Description

If 1, this define disables POR circuit in VLLS0, reducing power consumption even more. This define only has meaning if cPWR_DCDC_InBypass is TRUE.

Location

PWR_Configurations.h

Name

```
#define gAllowDeviceToSleep_c   0
```

Description

Functions PWR_AllowDeviceToSleep/PWR_DisallowDeviceToSleep simply decrement / increment a global variable. When variable value is 0, the system is allowed to enter low power. This define sets the initial value of this variable.

Location

PWR_Interface.h

Data type definitions

Name

```
typedef  union {
  uint8_t AllBits;
  struct {
    uint8_t   FromReset        :1;
    uint8_t   FromPSwitch_UART :1;
    uint8_t   FromKeyBoard     :1;
    uint8_t   FromLPTMR        :1;
    uint8_t   FromRadio        :1;
    uint8_t   FromBLE_LLTimer  :1;
    uint8_t   DeepSleepTimeout :1;
    uint8_t   SleepTimeout     :1;
  } Bits;
} PWRLib_WakeupReason_t;
```

Description

Data type above defines a bitmap created to identify the wakeup source. The function PWR_EnterLowPower() returns a value of this type for the low power modes other than 4 and 5 (the system exits these modes through reset sequence). A global variable of this type (extern volatile PWRLib_WakeupReason_t PWRLib_MCU_WakeupReason;) is also updated at wakeup. For mode 4 and 5 the global variable is updated at PWR_Init call. In case that system exits low power by a timer (FromLPTMR =1 or FromBLE_LLTimer=1) the DeepSleepTimeout is also set. SleepTimeout is set when the system exits from sleep (the system couldn't manage to enter deep sleep because a non low power timer was running).

Location

PWR_Interface.h

Name

```
typedef void ( *pfPWRCallBack_t ) ( void);
```

Description

Data type above defines a pointer to a callback function which is going to be called from the low power module. Functions of this type are registered in the low power module using two functions described later in the primitives section.

Location

PWR_Interface.h

API Primitives

Prototype

```
void PWR_Init( void);
```

Description

This function initializes all hardware modules involved in low power functionality. It must be called prior to any other function in the module.

Parameters

None

Returns

None

Prototype

```
bool_t PWR_ChangeDeepSleepMode(uint8_t dsMode);
```

Description

Call this function to change the deep sleep mode at run time.

Parameters

uint8_t dsMode: New deep sleep mode to be set. Valid values are from 0, 1, 2, 3, and 4. If 0 is choose the system doesn't enter deep sleep at all.

Returns

The function returns FALSE if it receive as parameter a value above four, and TRUE otherwise.

Prototype

```
uint8_t PWR_GetDeepSleepMode(void);
```

Description

Call this function to get the current deep sleep mode.

Parameters

None

Returns

The function returns the current value of deep sleep mode.

Prototype

```
void PWR_AllowDeviceToSleep( void );

void PWR_DisallowDeviceToSleep( void);
```

Description

The low power module maintains a global variable that enables/prevents the system to enter deep sleep. The system is allowed to enter deep sleep only when this variable is zero. Every software layer/entity that needs to keep awake the system calls PWR_DisallowDeviceToSleep and the variable is incremented. As software layers/entities decide that can enter deep sleep, they call

PWR_AllowDeviceToSleep and the variable is decremented. Software layers/entities must take care not to call PWR_AllowDeviceToSleep more times than PWR_DisallowDeviceToSleep.

Parameters

None

Returns

None

Prototype

```
bool_t PWR_CheckIfDeviceCanGoToSleep(void);
```

Description

The function can be used to check if the system is allowed to enter deep sleep or not.

Parameters

None

Returns

The function returns TRUE if the system is allowed to enter deep sleep and false otherwise.

Prototype

```
PWRLib_WakeupReason_t PWR_EnterLowPower( void );
```

Description

As stated before, a call to this function must be placed in the idle task to put the system in low power. First, this function checks if there are any non low power timers started. If there are, the function tries to put the system to sleep, otherwise to deep sleep. Next step is to check if the system is allowed to enter sleep/deep sleep by calling PWR_CheckIfDeviceCanGoToSleep. If the system is allowed to enter sleep/deep sleep more protocol specific checks are performed depending on the value of the cPWR_BLE_LL_Enable and cPWR_Zigbee_Enable defines. If everything is ok the system is finally put into sleep/deep sleep.

Parameters

None

Returns

If the system enters sleep, the function returns SleepTimeout field set in the PWRLib_WakeupReason_t bitmap. If the system enters deep sleep and the deep sleep mode is other than four, the function returns the wakeup source bitmap. If the wakeup source is a timer the DeepSleepTimeout field of the bitmap is also set. If any interrupt occurs during function execution, the function fails to put the system in deep sleep and returns 0 for all bitmap fields. If the deep sleep mode is 4 and the function successfully puts the system in deep sleep, the system exits deep sleep through reset sequence, so the function can’t return anything.

Prototype

```
void PWR_BLE_EnterDSM(uint16_t wakeupInstant);
```

Description

The function puts the BLE link layer in DSM immediately if it isn't already in this state. If it is, the function takes no actions. First the function sets the wakeup instant received as parameter in BLE link layer and then commands it to enter DSM. The function has meaning only if cPWR_BLE_LL_Enable is TRUE, otherwise it is empty.

Parameters

uint16_t wakeupInstant: the parameter represents the wakeup moment in regard with BLE link layer reference clock register (actually it wakes up earlier depending on the value of cPWR_BLE_LL_OffsetToWakeupInstant and cPWR_BLE_LL_OscStartupDelay defines). It works like a compare value. When the BLE link layer reference clock register reaches this value the BLE link layer wakes up and, if the MCU is in deep sleep also, it wakes up the entire SoC.

Returns

None

Prototype

```
uint16_t PWR_BLE_GetReferenceClock(void);
```

Description

The function reads the BLE link layer reference clock register. The function has meaning only if cPWR_BLE_LL_Enable is TRUE.

Parameters

None

Returns

The function returns the current value of the BLE link layer reference clock register. If cPWR_BLE_LL_Enable is FALSE the function returns 0.

Prototype

```
void PWR_BLE_ExitDSM(void);
```

Description

The function gets the BLE link layer out of DSM immediately. The function has meaning only if cPWR_BLE_LL_Enable is TRUE, otherwise it is empty.

Parameters

None

Returns

None

Prototype

```
void PWR_SetDeepSleepTimeInMs(  uint32_t deepSleepTimeTimeMs);
```

Description

The function sets the value of deep sleep duration. The function has meaning only for deep sleep mode 2 and 3.

Parameters

uint32_t deepSleepTimeTimeMs: The new value of deep sleep duration. Upon entering deep sleep the value is truncated in regard with the maximum deep sleep duration possible in each mode.

Returns

None

Prototype

```
void PWR_RegisterLowPowerEnterCallback( pfPWRCallBack_t lowPowerEnterCallback);
```

Description

This function registers in the low power module a function which will be called just before entering low power modes 1,2,3,4, and 5. The callback is mainly intended to call the DCDC function DCDC_PrepareForPulsedMode but other low power settings (LEDs off and other GPIO configurations to minimize power consumption) can be made here as well. DCDC_PrepareForPulsedMode should be the last function appealed from this callback.

Parameters

pfPWRCallBack_t lowPowerEnterCallback: The function to be called by the low power module just before entering low power.

Returns

None

Prototype

```
void PWR_RegisterLowPowerExitCallback( pfPWRCallBack_t lowPowerExitCallback);
```

Description

This function registers in the low power module a function which will be called just after exiting low power modes 1,2,3,4, and 5 (for modes 4 and 5 it is called just in the case the system fails to enter low power. Otherwise, the system exits from these modes through reset sequence). The callback is mainly intended to call the DCDC function DCDC_PrepareForContinuousMode but other run mode settings (get back to run mode settings for LEDs and other GPIO, etc) can be made here as well. DCDC_PrepareForContinuousMode should be the first function appealed from this callback.

Parameters

pfPWRCallBack_t lowPowerExitCallback: The function to be called by the low power module just after exiting low power.

Returns

None

4. Drivers

4.1.1. Overview

A set of high-level drivers for LEDs and keyboard is provided, but it is 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 enables control of up to four 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 four).

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

TurnOnLeds ()

Prototype:

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

Description:
Turns on all LEDs.

Parameters:

None.

Returns:

None.

SerialFlasing ()

Prototype:

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

Description:
Turns on serial flashing on all LEDs.

Parameters:

None.

Returns:

None.

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.

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.

LED_Init ()

Prototype:

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

Description:
Initializes the LED module.

Parameters:

None.

Returns:

None.

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.

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.

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.

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.

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.

LED_TurnOffAllLeds ()

Prototype:

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

Description:
Turns off all the LEDs.

Parameters:

None.

Returns:

None.

LED_TurnOnAllLeds ()

Prototype:

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

Description:
Turns on all the LEDs.

Parameters:

None.

Returns:

None.

LED_StopFlashingAllLeds ()

Prototype:

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

Description:
Stops flashing and turns off all LEDs.

Parameters:

None.

Returns:

None.

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.

LED_StartBlip ()

Prototype:

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

Description:

Sets up one or more LEDs for blinking at once.

Parameters:

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

Returns:

None.

LED_StopFlash ()

Prototype:

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

Description:

Stops an LED from flashing.

Parameters:

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

Returns:

None.

LED_StartSerialFlash ()

Prototype:

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

Description:

Starts serial flashing LEDs.

Parameters:

None.

Returns:

None.

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.

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 enables controlling of up to four 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 / long 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 milliseconds 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 provided 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 provided 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

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 the application callback function.

Returns:

None.

KBD_IsWakeupSource

Prototype:

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

Description:
Indicates whether 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 enables the 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 number 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

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 the 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.

GpioUninstallIsr

Prototype:

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

Description:
Uninstalls the ISR callback for the specified GPIO.

Parameters:

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

Returns:

The error code.

4.5. Kinetis MKW40Z DCDC Driver Reference

4.5.1. Overview

The DCDC converter module is a switching mode DC-DC converter supporting Buck, Boost, and Bypass mode. The DCDC converter produces two switching outputs in both the Buck and Boost modes. They are VDD1p45 and VDD1p8, which can produce up to 25mA and 42.5mA continuous output current respectively.

In Buck mode, it supports operation with battery voltage from 2.1V to 4.2V.

In Boost mode, it supports operation with battery voltage from 0.9V to 1.8V.

In Bypass mode, the DCDC converter is disabled. Individual supply signals of KW40 need to be supplied with regulated supply accordingly.

The DCDC operating mode is hardware selected through configuration jumpers.

The DCDC driver purpose is to help the DCDC to operate properly.

At DCDC driver initialization (DCDC_Init) the user passes to the driver DCDC operating parameters: battery voltage range, DCDC operating mode, the time interval in milliseconds at which the driver should monitor and improve DCDC functionality, a callback to notify the application about the DCDC status, and the desired output targets of the DCDC power lines.

If the DCDC operating mode chosen is bypass the driver takes no further actions. Otherwise, the driver starts a low power interval timer (the timer interval is the one received in the configuration structure at DCDC_Init) in order to monitor and improve DCDC operation periodically. Each time the low power timer expires, the driver measures the battery voltage using ADC driver and sets its value in DCDC hardware, determines the output targets for the power lines according to the battery voltage and user desired values, sets those values in the appropriate registers and then calls the application callback to notify the user about the DCDC status.

For example, if the DCDC operates in boost mode, the measured battery voltage is 1.5 V and the output target values requested by the user are 1.8 V for the 1.8 V power line and 1.45 V for the 1.45 V power line, the output target values determined by the driver are 1.8 V for the 1.8 V power line and 1.55 V for the 1.45 V power line since in Boost mode the output voltages must be at least 50 mV above the battery voltage.

4.5.2. Application Programming Interface

DCDC_Init() function must be called prior to any other DCDC function. The function calls TMR_Init() function from Timers Manager to ensure timers are initialised (DCDC_init function starts a low power interval timer. TMR_Init() function only runs the first time it is called.).

The DCDC driver initializes and uses the ADC driver. This will generate contention if other parts of the program also use the ADC driver or the ADC module.

Configuration Macros

Name:

```
#define gDCDC_Enabled_d                     0
```

Description:

Enables/Disables DCDC related code and variables.

Location:

DCDC.h

Data type definitions

Name:

```
typedef enum
{
  gDCDC_PSwStatus_Low_c,
  gDCDC_PSwStatus_High_c
}dcdcPSwStatus_t;
```

Description:

Defines the status of the power switch used to start the DCDC in buck mode.

Location:

DCDC.h

Name:

```
typedef enum
{
  gDCDC_PSwIntEdge_Rising_c = 1,
  gDCDC_PSwIntEdge_Falling_c,
  gDCDC_PSwIntEdge_All_c
}dcdcPSwIntEdge_t;
```

Description:

Defines the edges of power switch input that can be chosen to generate an interrupt.

Location:

DCDC.h

Name:

```
typedef void (*pfDCDCPSwitchCallback_t)(dcdcPSwStatus_t);
```

Description:

Defines the callback function type that could be installed to be called from the power switch interrupt.

Location:

DCDC.h

Name:

```
typedef enum
{
  gDCDC_Mode_Bypass_c,
  gDCDC_Mode_Buck_c,
  gDCDC_Mode_Boost_c,
}dcdcMode_t;
```

Description:

Defines the DCDC operation modes. The DCDC operation mode is hardware configured through jumpers but the DCDC driver must be noticed about the chosen configuration.

Location:

DCDC.h

Name:

```
typedef enum
{
  gDCDC_Event_NoEvent_c = 0x0,
  gDCDC_Event_VBatOutOfRange_c = 0x1,
  gDCDC_Event_1P45OutputTargetChange_c = 0x2,
  gDCDC_Event_1P8OutputTargetChange_c = 0x4
}dcdcEvent_t;
```

Description:

Defines the events DCDC driver notifies the application about:

- gDCDC_Event_VBatOutOfRange_c: VDCDC_IN (battery voltage) is no longer in the range defined by the user in the DCDC configuration structure passed as parameter to the DCDC_Init function.
- gDCDC_Event_1P45OutputTargetChange_c: 1.45 power line output target value changed since the last application notification.
- gDCDC_Event_1P8OutputTargetChange_c: 1.8 power line output target value changed since the last application notification.

Location:

DCDC.h

Name:

```
typedef enum
{
  gDCDC_1P45OutputTargetVal_1_275_c = 0,
  gDCDC_1P45OutputTargetVal_1_300_c,
  gDCDC_1P45OutputTargetVal_1_325_c,
  gDCDC_1P45OutputTargetVal_1_350_c,
  gDCDC_1P45OutputTargetVal_1_375_c,
  gDCDC_1P45OutputTargetVal_1_400_c,
  gDCDC_1P45OutputTargetVal_1_425_c,
  gDCDC_1P45OutputTargetVal_1_450_c,
  gDCDC_1P45OutputTargetVal_1_475_c,
  gDCDC_1P45OutputTargetVal_1_500_c,
  gDCDC_1P45OutputTargetVal_1_525_c,
  gDCDC_1P45OutputTargetVal_1_550_c,
  gDCDC_1P45OutputTargetVal_1_575_c,
  gDCDC_1P45OutputTargetVal_1_600_c,
  gDCDC_1P45OutputTargetVal_1_625_c,
  gDCDC_1P45OutputTargetVal_1_650_c,
  gDCDC_1P45OutputTargetVal_1_675_c,
  gDCDC_1P45OutputTargetVal_1_700_c,
  gDCDC_1P45OutputTargetVal_1_725_c,
  gDCDC_1P45OutputTargetVal_1_750_c,
  gDCDC_1P45OutputTargetVal_1_775_c,
  gDCDC_1P45OutputTargetVal_1_800_c
}dcdc1P45OutputTargetVal_t;
```

Description:

Defines the output target values for the DCDC 1,45V power line. In buck mode this value is limited at 1,65V (gDCDC_1P45OutputTargetVal_1_650_c).

Location:

DCDC.h

Name:

```
typedef enum
{
  gDCDC_1P8OutputTargetVal_1_650_c = 0,
  gDCDC_1P8OutputTargetVal_1_675_c,
  gDCDC_1P8OutputTargetVal_1_700_c,
  gDCDC_1P8OutputTargetVal_1_725_c,
  gDCDC_1P8OutputTargetVal_1_750_c,
  gDCDC_1P8OutputTargetVal_1_775_c,
  gDCDC_1P8OutputTargetVal_1_800_c,
  gDCDC_1P8OutputTargetVal_1_825_c,
  gDCDC_1P8OutputTargetVal_1_850_c,
  gDCDC_1P8OutputTargetVal_1_875_c,
  gDCDC_1P8OutputTargetVal_1_900_c,
  gDCDC_1P8OutputTargetVal_1_925_c,
  gDCDC_1P8OutputTargetVal_1_950_c,
  gDCDC_1P8OutputTargetVal_1_975_c,
  gDCDC_1P8OutputTargetVal_2_000_c,
  gDCDC_1P8OutputTargetVal_2_025_c,
  gDCDC_1P8OutputTargetVal_2_050_c,
  gDCDC_1P8OutputTargetVal_2_800_c = 0x20,
  gDCDC_1P8OutputTargetVal_2_825_c,
  gDCDC_1P8OutputTargetVal_2_850_c,
  gDCDC_1P8OutputTargetVal_2_875_c,
  gDCDC_1P8OutputTargetVal_2_900_c,
  gDCDC_1P8OutputTargetVal_2_925_c,
  gDCDC_1P8OutputTargetVal_2_950_c,
  gDCDC_1P8OutputTargetVal_2_975_c,
  gDCDC_1P8OutputTargetVal_3_000_c,
  gDCDC_1P8OutputTargetVal_3_025_c,
  gDCDC_1P8OutputTargetVal_3_050_c,
  gDCDC_1P8OutputTargetVal_3_075_c,
  gDCDC_1P8OutputTargetVal_3_100_c,
  gDCDC_1P8OutputTargetVal_3_125_c,
  gDCDC_1P8OutputTargetVal_3_150_c,
  gDCDC_1P8OutputTargetVal_3_175_c,
  gDCDC_1P8OutputTargetVal_3_200_c,
  gDCDC_1P8OutputTargetVal_3_225_c,
```

```
  gDCDC_1P8OutputTargetVal_3_250_c,
  gDCDC_1P8OutputTargetVal_3_275_c,
  gDCDC_1P8OutputTargetVal_3_300_c,
  gDCDC_1P8OutputTargetVal_3_325_c,
  gDCDC_1P8OutputTargetVal_3_350_c,
  gDCDC_1P8OutputTargetVal_3_375_c,
  gDCDC_1P8OutputTargetVal_3_400_c,
  gDCDC_1P8OutputTargetVal_3_425_c,
  gDCDC_1P8OutputTargetVal_3_450_c,
  gDCDC_1P8OutputTargetVal_3_475_c,
  gDCDC_1P8OutputTargetVal_3_500_c,
  gDCDC_1P8OutputTargetVal_3_525_c,
  gDCDC_1P8OutputTargetVal_3_550_c,
  gDCDC_1P8OutputTargetVal_3_575_c
 }dcdc1P8OutputTargetVal_t;
```

Description:

Defines the output target values for the DCDC 1,8V power line.

Location:

DCDC.h

Name:

```
typedef struct dcdcCalbackParam_tag
{
  dcdcEvent_t dcdcEvent;
  dcdc1P45OutputTargetVal_t dcdc1P45OutputTargetVal;
  dcdc1P8OutputTargetVal_t dcdc1P8OutputTargetVal;
  uint16_t dcdc1P8OutputMeasuredVal;
  uint16_t dcdcVbatMeasuredVal;
}dcdcCallbackParam_t;
```

Description:

Defines the parameter type the application callback will provide to the user. The structure contains a notification event, the current output target values for the 1.45 and 1.8 power lines, the measured value in mV of the 1.8V power line and the measured value in mV of the VDCDC_IN (battery voltage).

Location:

DCDC.h

Name:

```
typedef void (*pfDCDCAppCallback_t)(const dcdcCallbackParam_t*);
```

Description:

Defines application callback type. The application callback is passed to the DCDC driver in the DCDC configuration structure passed as parameter to the DCDC_Init function.

Location:

DCDC.h

Name:

```
typedef struct dcdcConfig_tag
{
  uint16_t vbatMin;
  uint16_t vbatMax;
  dcdcMode_t dcdcMode;
  uint32_t vBatMonitorIntervalMs;
  pfDCDCAppCallback_t pfDCDCAppCallback;
  dcdc1P45OutputTargetVal_t dcdc1P45OutputTargetVal;
  dcdc1P8OutputTargetVal_t  dcdc1P8OutputTargetVal;
}dcdcConfig_t;
```

Description:

Defines the DCDC configuration structure type. A pointer to a DCDC configuration structure is passed as parameter to the DCDC_Init function. The structure contains the operating range for the VDCDC_IN (battery voltage), the hardware configured DCDC mode of operation, the time interval in milliseconds at which the VDCDC_IN (battery voltage) will be monitored and application callback will be called, the application callback address, and the desired output target values for the 1.45V and 1.8V power lines.

Location:

DCDC.h

API Primitives

Prototype

```
bool_t DCDC_Init( const dcdcConfig_t * pDCDCConfig);
```

Description

First, the function verifies the DCDC configuration structure parameters and, if they are valid, it keeps internally a pointer to the structure. If the DCDC mode in the configuration structure is bypass, the function takes no other actions. Otherwise, the function initializes the ADC driver, makes the appropriate DCDC settings and initiates the monitoring procedure by starting an interval low power timer.

Parameters

const dcdcConfig_t * pDCDCConfig. Pointer to the DCDC configuration structure. The configuration structure lifetime must be the entire program execution.

Returns

FALSE: if one of the parameters is not valid, ADC driver initialization failed, low power interval timer could not be started.

TRUE: if all initialization steps succeeded.

Prototype

```
bool_t DCDC_SetOutputVoltageTargets
(
dcdc1P45OutputTargetVal_t dcdc1P45OutputTargetVal,
dcdc1P8OutputTargetVal_t  dcdc1P8OutputTargetVal
);
```

Description

The function gives the user the possibility to change the desired output target voltages at run time. The values that will be set in DCDC registers will be the closest possible values to the desired ones in respect with the current battery voltage value. The desired values are kept internally in the driver and will be automatically set in the DCDC registers when the battery voltage value will allow it. If this function isn't called the driver will use the output target values from the DCDC configuration structure.

Parameters

dcdc1P45OutputTargetVal_t dcdc1P45OutputTargetVal: the desired output target voltage level for the 1.45V power line.

dcdc1P8OutputTargetVal_t dcdc1P8OutputTargetVal: the desired output target voltage level for the 1.8V power line.

Returns

FALSE: if one of the parameters is not valid or if this function is called prior to DCDC_Init.

TRUE: if executes successfully.

Prototype

```
uint16_t DCDC_1P45OutputTargetTomV(dcdc1P45OutputTargetVal_t dcdc1P45OutputTarget);
```

Description

The function converts dcdc1P45OutputTargetVal_t to millivolts.

Parameters

dcdc1P45OutputTargetVal_t dcdc1P45OutputTarget: the dcdc1P45OutputTargetVal_t constant to be converted in millivolts.

Returns

The conversion result as uint16_t.

Prototype

```
uint16_t DCDC_1P8OutputTargetTomV(dcdc1P8OutputTargetVal_t dcdc1P8OutputTarget);
```

Description

The function converts dcdc1P8OutputTargetVal_t to millivolts.

Parameters

dcdc1P8OutputTargetVal_t dcdc1P8OutputTarget: the dcdc1P8OutputTargetVal constant to be converted in millivolts.

Returns

The conversion result as uint16_t.

Prototype

```
bool_t DCDC_PrepareForPulsedMode(void);
```

Description

The function makes the pulsed mode appropriate settings in the DCDC hardware. In connectivity software system the DCDC works in pulsed mode when the system enters low power (LLS, VLLS). This function should be called just before putting the system in low power if the DCDC operating mode is boost or buck.

Parameters

None

Returns

FALSE if this function is called prior to DCDC_Init.

TRUE otherwise.

Prototype

```
bool_t DCDC_PrepareForContinuousMode(void);
```

Description

The function restores continuous mode settings in the DCDC hardware. In connectivity software system the DCDC works in continuous mode except when the system enters low power (LLS, VLLS). This function should be called just after the system exits low power if the DCDC operates in boost or buck and DCDC_PrepareForPulsedMode was called before entering low power.

Parameters

None

Returns

FALSE if this function is called prior to DCDC_Init.

TRUE otherwise.

Prototype

```
bool_t DCDC_SetUpperLimitDutyCycle(uint8_t upperLimitDutyCycle);
```

Description

The function limits the DCDC converter duty cycle in buck or boost mode.

Parameters

uint8_t upperLimitDutyCycle the maximum duty cycle to be set in the hardware.

Returns

FALSE if this function is called prior to DCDC_Init or if the parameter is invalid.

TRUE if the maximum duty cycle is set in hardware or if the DCDC is operating in bypass mode.

Prototype

```
bool_t DCDC_GetPowerSwitchStatus(dcdcPSwStatus_t* pDCDCPSwStatus);
```

Description

The function sets the variable referenced by its parameter to the current value of the DCDC power switch status.

Parameters

dcdcPSwStatus_t* pDCDCPSwStatus pointer to the variable to be set at DCDC power switch status.

Returns

FALSE if this function is called prior to DCDC_Init.

TRUE if the variable value is set at DCDC power switch status.

Prototype

```
void DCDC_ShutDown(void);
```

Description

The function stops the DCDC if it operates in buck mode. If the function is called prior to DCDC_Init or if the DCDC operating mode is not buck, the system returns from this function. Otherwise the system will be shut down.

Parameters

None.

Returns

None

Prototype

```
bool_t DCDC_InstallPSwitchCallback(pfDCDCPSwitchCallback_t pfPSwCallback, dcdcPSwIntEdge_t
pSwIntEdge);
```

Description

The function behavior depends on the pfPSwCallback value:

If pfPSwCallback is not NULL, The function enables the DCDC interrupt in DCDC hardware and NVIC and installs pfPSwCallback to be called when DCDC power switch interrupt occurs.

If pfPSwCallback is NULL the function disables the DCDC interrupt in DCDC hardware and NVIC.

Be aware that the callback is called directly from the DCDC interrupt.

Parameters

pfDCDCPSwitchCallback_t pfPSwCallback: the callback to be called when DCDC power switch interrupt occurs.

dcdcPSwIntEdge_t pSwIntEdge: selects the edge on which the DCDC power switch interrupt occurs.

Returns

FALSE if this function is called prior to DCDC_Init.

TRUE otherwise.

5. Revision history

The following table summarizes the changes done to the document since the initial release.

Table 3. Revision history

Revision	Date	Substantive changes
0	06/2015	Initial release.
1	09/2015	Added KW40Z Low Power, DCDC and production data storage

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number: CONNFWKRM
Rev. 1
09/2015