NXP Semiconductors
User's Guide

Document identifier: CONNFWRM
Rev. 13, 01 June 2020

Connectivity Framework Reference Manual

Chapter 1
Introduction

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

1.2 References

- http://www.nxp.com/kinetis
- www.arm.com/cortex-microcontroller-software-interface-standard

1.3 Acronyms and abbreviations

Table 1. Acronyms and abbreviations

Acronym / term	Description
FSCI	Framework Serial Communication Interface
NV Storage	Non-Volatile Storage
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

Chapter 2
Overview

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

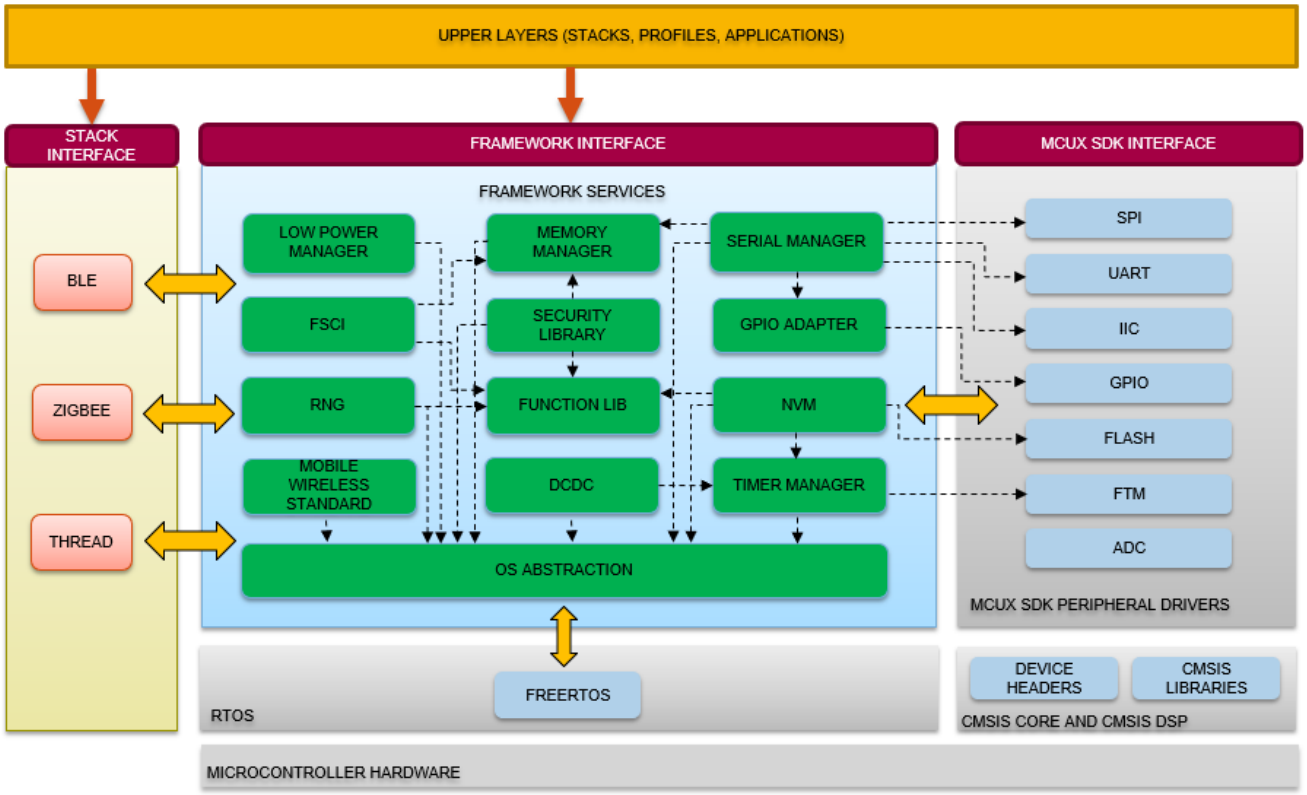


Figure 1. System decomposition

All 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.

Chapter 3
Framework Services

3.1 OS abstraction

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, which ensures portability across multiple operating systems. If the use of an operating system which is not currently supported by the OS Abstraction layer is desired, an OS adapter must be implemented.

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

To add support for another operating system, all functions and macros described in this chapter must be implemented for each RTOS. In addition, all typedefs should 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, some restrictions and enhancements are needed within the OS Abstraction layer that are reflected throughout code.

Currently, OS Abstraction layers are implemented for FreeRTOS OS and bare metal. The bare metal abstraction layer offers a non-preemptive, priority-based, task scheduler.

3.1.1 Connectivity Framework Bare Metal Task Scheduler

The connectivity framework bare metal task scheduler has been developed to have a non-preemptive, priority-based scheduler, which is used to conceptually separate various portions of protocol stacks.

The MCU interrupts operate independently of tasks and may often pass control to a task by using events.

Each task must have its own event handler, as shown in the following example. The initialization code for a task is optional. Each event is a single bit in an event bit mask and is defined by the task. Multiple events may be set at the same time.

```
void MyTask(uint32_t parameter)
{
    osaEventFlags_t ev;
    static uint8_t initialized = FALSE;

    if(!initialized)
    {
        /* place initialization code here... */
        myEventId = OSA_EventCreate(TRUE);
        initialized = TRUE;
    }

    while(1)
    {
        /* place event handler code here... */
        OSA_EventWait(myEventId, 0x00FFFFFF, FALSE, osaWaitForever_c, &ev);

        if (ev & myEvent_1)
        {
            /* place event handler code here... */

        }

        if(!gUseRtos_c)
        {
```

```
            break;
        }
    }
}
```

Some functions use a combination of an event bit and a message queue to communicate data.

Tasks are **non-preemptive**, which means that, once a task gains control, it has full control until it completes or returns from the task event handler function. Tasks should complete in less than 2 ms to avoid starving other tasks of processing time. If the task takes too long to process, it prevents the stack's lower layers from processing incoming packets. Under no circumstance should a task take longer than 10 ms.

The *OSA_Init()* and *OSA_Start()* API are called from main() to initialize and start the bare metal scheduler. For more information, see the *fsl_os_abstraction_bm.c* file.

```
int main (void)
{
    OSA_Init();
    /* Initialize MCU clock */
    hardware_init();
    OSA_TimeInit();
    OSA_TaskCreate(OSA_TASK(main_task),NULL);
    OSA_Start();
    return 0;}
```

3.1.1.1 Internal functionality

The entire functionality of the framework bare metal scheduler is implemented in the "*fsl_os_abstraction_bm.c*" file, which can be found at the following location: *<SDK_Path\middleware\wireless\framework\OSAbstraction\Source*

Once a task is created, it is inserted into a linked list, which is ordered by priority. The task is placed in a "Ready" state.

The *TASK_MAX_NUM* macro specifies the maximum number of tasks that can be created.

The scheduler iterates through the list searching for the highest priority task which needs to run. After the task has run, the scheduler restarts the search from the highest priority task.

NOTE

If a task of a certain priority does not block on an event, it does not allow other lower priority tasks to run.

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. The user must implement a function with the prototype *extern void main_task(void *)* and treat it like a task. The OS Abstraction implementation declares this function external.

From this task, you can create other tasks, previously defined with *OSA_TASK_DEFINE(name, priority, instances, stackSz, useFloat)*. After system initialization, the *main_task* can either be terminated or reused. Note that terminating *main_task* does not necessarily free the used memory because the task stack is defined as a global array for some RTOSes.

The *main_task* initially has the lowest priority. If necessary, the priority can be modified at runtime, using the *OSA_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_TASK_DEFINE* macro at compile time and are not automatically started. After that, tasks can be created anytime by using the *OSA_TaskCreate* API.

For FreeRTOS OS, 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.

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:

OSA defines sixteen priority levels and names some of them.

Name:

```
#define OSA_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_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 osaStatus_tag
{
    osaStatus_Success = 0U,
    osaStatus_Error = 1U,
    osaStatus_Timeout = 2U,
    osaStatus_Idle = 3U
}osaStatus_t;
```

Description:

OSA error codes.

Name:

```
typedef void* osaTaskParam_t;
```

Description:

The data type definition for the parameter the task receives at creation.

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(void* param);
```

Description: Prototype of the user-implemented *main_task*.

Parameters:

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

Returns:

None.

OSA_ TaskCreate ()

Prototype:

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

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

Parameters:

Name	Type	Direction	Description
thred_def	osaThreadDef_t*	[IN]	Pointer to the task definition.
task_param	osaTaskParam_t	[IN]	Parameter to pass to the newly created task.

Returns:

Thread ID for reference by other functions. If an error occurs, the ID is NULL.

OSA_TaskGetId ()

Prototype:

```
osaTaskId_t OSA_TaskGetId(void);
```

Description: Returns the thread ID of the calling thread.

Parameters:

None.

Returns:

ID of the calling thread.

OSA _TaskDestroy ()

Prototype:

```
osaStatus_t OSA_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 an invalid parameter.

OSA_TaskYield ()

Prototype:

```
osaStatus_t OSA_TaskYield(void);
```

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

Parameters:

None.

Returns:

- osaStatus_Success – The function is called successfully.
- osaStatus_Error – An error occurs.

OSA_TaskSetPriority ()

Prototype:

```
osaStatus_t OSA_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_ TaskGetPriority ()

Prototype:

```
osaTaskPriority_t OSA_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:

The current priority value of the thread.

OSA_ TimeDelay ()

Prototype:

```
void OSA_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.

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

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

    OSA_TaskDestroy (OSA_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 similar for all operating systems, except for the allocation procedure. For bare metal, the semaphores are allocated within the OS Abstraction layer, while FreeRTOS OS allocates them internally. For bare metal there is also no context switch when the semaphore can't be taken, so the function status must be checked.

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 a semaphore ID.

3.1.3.4 API primitives

OSA_SemaphoreCreate ()

Prototype:

```
osaSemaphoreId_t OSA_SemaphoreCreate(uint32_t initValue);
```

Description: Creates and initializes a semaphore object used for managing resources.

Parameters:

Name	Type	Direction	Description
initValue	uint32_t	[IN]	Initial semaphore count.

Returns:

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

OSA_SemaphoreDestroy ()

Prototype:

```
osaStatus_t OSA_SemaphoreDestroy(osaSemaphoreId_t semId);
```

Description: Destroys the semaphore object and frees used memory.

Parameters:

Name	Type	Direction	Description
semId	osaSemaphoreId_t	[IN]	Semaphore ID returned by OSA_SemaphoreCreate.

Returns:

- osaStatus_Success – The semaphore is successfully destroyed.
- osaStatus_Error – The semaphore cannot be destroyed.

OSA_SemaphoreWait ()

Prototype:

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

Description: Takes a semaphore token.

Parameters:

Name	Type	Direction	Description
semId	osaSemaphoreId_t	[IN]	Semaphore ID returned by OSA_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_SemaphorePost ()

Prototype:

osaStatus_t OSA_SemaphorePost(osaSemaphoreId_t semId);

Description: Releases a semaphore token.

Parameters:

Name	Type	Direction	Description
semId	osaSemaphoreId_t	[IN]	Semaphore ID returned by OSA_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 FreeRTOS OS only, mutexes are implemented with the priority inheritance mechanism. For bare metal the mutexes are allocated within the OS Abstraction layer, while FreeRTOS OS allocates them internally. For bare metal there is also no context switch when the mutex cannot be taken, so the function status must be checked.

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.

3.1.5 Message queues

3.1.5.1 Overview

For bare metal, the OS Abstraction layer allocates the memory for the queues, but, for FreeRTOS OS, the queue allocation is done by the OS. Messages are defined to be single 32-bit values or pointers. For bare metal there is also no context switch when a task waits for a message, so the function status must be checked to decide whether a message has arrived or not.

3.1.5.2 Constant macro definitions

Name:

```
#define osNumberOfMessageQs 5
```

Description:

Defines the maximum number of message queues.

Name:

```
#define osNumberOfMessages 40
```

Description:

Defines the maximum number of messages in a message queue.

3.1.5.3 User-defined data type definitions

Name:

```
typedef void* osaMsgQId_t;
```

Description:

Data type definition for a message queue ID.

Name:

```
typedef void* osaMsg_t;
```

Description:

Data type definition for a queue message type.

3.1.5.4 API primitives

OSA_MsgQCreate ()

Prototype:

```
osaMsgQId_t OSA_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 accommodate

Returns:

Message queue handle if successful, or NULL if failed.

OSA_MsgQDestroy ()

Prototype:

```
osaStatus_t OSA_MsgQDestroy(osaMsgQId_t msgQId);
```

Description: Destroys a message queue.

Parameters:

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

Returns:

- osaStatus_Success – The queue was successfully destroyed.
- osaStatus_Error – The message queue destruction failed.

OSA_MsgQPut ()

Prototype:

```
osaStatus_t OSA_MsgQPut(osaMsgQId_t msgQId, void* pMessage);
```

Description: Puts a message into the message queue.

Parameters:

Name	Type	Direction	Description
msgQId	osaMsgQId_t	[IN]	The queue handler returned by *OSA_MsgQCreate()*
pMessage	Void*	[IN]	The address of the message to be queued

Returns:

- osaStatus_Success – A message is successfully put into the queue.
- osaStatus_Error – The queue is full or an invalid parameter is passed.

OSA_MsgQGet ()

Prototype:

```
osaStatus_t OSA_MsgQGet(osaMsgQId_t msgQId, void *pMessage, uint32_t millisec);
```

Description: Gets a message from the message queue.

Parameters:

Name	Type	Direction	Description
msgQId	osaMsgQId_t	[IN]	The queue handler returned by *OSA_MsgQCreate()*
pMessage	void *	[IN]	The address where the message will be received.
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 – An invalid parameter.

3.1.6 Events

3.1.6.1 Overview

When waiting for events, a mask of events is passed to the API, which indicates the desired event flags to wait for. A predefined mask *osaEventFlagsAll_c* can be used to wait for any of the possible events. The waitAll parameter establishes whether to wait for at least one of the events in the mask (waitAll = 0) or for all of them (waitAll =1). For bare metal there is no context switch when a task waits for an event, so the function status must be checked to decide whether an event has occurred.

3.1.6.2 Constant macro definitions

Name:

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

Description:

The number of event objects.

Name:

```
#define osaEventFlagsAll_c ((osaEventFlags_t)(0x00FFFFFF))
```

Description:

A mask representing all possible event flags.

3.1.6.3 User-defined data type definitions

Name:

```
typedef void* osaEventId_t;
```

Description:

Data type definition for the event objects, which is a pointer to a pool of objects.

Name:

```
typedef uint32_t osaEventFlags_t;
```

Description:

Data type definition for the event flags group.

3.1.6.4 API primitives

OSA_EventCreate ()

Prototype:

```
osaEventId_t OSA_EventCreate(bool_t autoClear);
```

Description: Creates event objects.

Parameters:

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

Returns:

Event ID if success, NULL if creation failed.

OSA_EventDestroy ()

Prototype:

```
osaStatus_t OSA_EventDestroy(osaEventId_t eventId);
```

Description: Destroys an event object.

Parameters:

Name	Type	Direction	Description
eventId	osaEventId_t	[IN]	The event handler returned by the OSA_EventCreate.

Returns:

- osaStatus_Success – The event is successfully destroyed.

- osaStatus_Error – The event destruction failed.

OSA_EventSet ()

Prototype:

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

Description: Sets the specified signal flags of an event object.

Parameters:

Name	Type	Direction	Description
eventId	osaEventId_t	[IN]	The event handler returned by the OSA_EventCreate.
flagsToSet	osaEventFlags_t	[IN]	flags to set

Returns:

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

OSA_EventClear ()

Prototype:

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

Description: Clears the specified event flags of an event object.

Parameters:

Name	Type	Direction	Description
eventId	osaEventId_t	[IN]	The event handler returned by the OSA_EventCreate.
flagsToClear	osaEventFlags_t	[IN]	flags to clear

Returns:

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

OSA_EventWait ()

Prototype:

```
osaStatus_t OSA_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 handler returned by the OSA_EventCreate.
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 osaWaitForever_c 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 Interrupts

OSA_InterruptDisable ()

Prototype:

```
void OSA_InterruptDisable(void);
```

Description: Disables all interrupts.

Parameters:

None.

Returns:

None.

OSA_InterruptEnable ()

Prototype:

```
void OSA_InterruptEnable(void);
```

Description: Enables all interrupts.

Parameters:

None.

Returns:

None.

OSA_InstallIntHandler ()

Prototype:

```
void * OSA_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 (*)(void)	[IN]	Handler function

Returns:

- If successful, returns the previous ISR handler .
- Returns NULL if the handler cannot be installed.

3.2 Message Management

3.2.1 Overview

A memory management module, which is organized into partitions of identical memory blocks, is included in the framework. 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. As a result, 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. The user 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. Semaphores, mutexes, signals, and so on, are provided for this purpose.

The user can send a message to a receiving task and then activate the synchronization element. The message is dequeued after waiting for the synchronization signal. The actual memory buffer is allocated by the sending task and must be freed or reused by the receiving task. In terms of memory space, this approach requires only an additional linked list anchor.

The messaging module is a blend of macros and functions on top of the Memory Manager and List 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 List 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 != NULL)
```

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; 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:

```
void* MSG_Alloc (uint32_t msgSize);
```

Description: Allocates a message.

Parameters:

Name	Type	Direction	Description
msgSize	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) MSG_Alloc(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:

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

Description: Frees a message.

Parameters:

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

Returns:

Status of the freeing operation.

MSG_FreeQueue

Prototype:

```
#define  MSG_FreeQueue(anchor)
while(MSG_Pending(anchor))
{ MSG_Free(MSG_DeQueue(anchor)); }
```

Description: Free memory of all queue elements.

Parameters:

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

Returns:

Status of the freeing operation.

MSG_GetSize

Prototype:

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

Description: This function queries the message's buffer size.

Parameters:

Name Type Direction Description buffer void* IN Pointer to the message buffer

Returns:

The message's buffer size.

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

The Connectivity Framework provides two memory managers.

- Primary allocator called *Legacy Memory Manager*
- Secondary allocator called *Light Memory Manager*

NOTE

The *Light Memory Manager* meets the low-power requirements. This new allocator is developed and tested on KW38, running Low-Power reference design applications. Do not use it on any other application or device.

3.3.1 Overview

A generic memory allocation scheme is undesirable due to the potential for memory fragmentation issues during use. Therefore, the framework provides a non-fragmenting memory allocation solution. The solution relies on partitions to avoid the memory fragmentation problem and 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. In other words, if no buffer of the requested size is available, the allocation routine returns

a larger buffer. As per requirements, the partition must be defined in the ascending size order and the block sizes must be multiples of four to ensure the block alignment to four bytes:

```
#ifndef PoolsDetails_c
#define PoolsDetails_c \
    _block_size_ 64 _number_of_blocks_ 8 _pool_id_(0) _eol_ \
    _block_size_ 128 _number_of_blocks_ 4 _pool_id_(0) _eol_ \
    _block_size_ 256 _number_of_blocks_ 6 _pool_id_(0) _eol_
#endif
```

For this example, three partitions must be created. In addition to the requested amount of memory, each block uses a header for internal bookkeeping. You must not use this header because some information is still needed for deallocation.

3.3.2 Constant macro definitions

Name:

```
#ifndef PoolsDetails_c
#define PoolsDetails_c \
    _block_size_ 64 _number_of_blocks_ 8 _pool_id_(0) _eol_ \
    _block_size_ 128 _number_of_blocks_ 2 _pool_id_(0) _eol_ \
    _block_size_ 256 _number_of_blocks_ 6 _pool_id_(0) _eol_
#endif
```

Description:

Defines the block size and number of blocks for every memory pool.

Name:

```
#ifndef MEM_CheckMemBufferThreshold_c
#define MEM_CheckMemBufferThreshold_c 120000 /* ms */
#endif
```

Description:

Defines the maximum allocation duration for a memory buffer for debugging.

This value is used by the MEM_CheckIfMemBuffersAreFreed() function.

Name:

```
#ifndef MEM_CheckMemBufferInterval_c
#define MEM_CheckMemBufferInterval_c 15000 /* ms */
#endif
```

Description:

Defines the interval at which the memory pools should be scanned for expired buffers for debugging.

This value is used by the MEM_CheckIfMemBuffersAreFreed() function.

Name:

```
#ifndef MEM_BufferAlloc
#define MEM_BufferAlloc(numBytes) MEM_BufferAllocWithId(numBytes, 0, (void*)__get_LR())
#endif
```

Description:

Defines the default memory allocation API which should be used by the application.

Name:

```
#define MEM_BufferAllocForever(numBytes,poolId) \
 MEM_BufferAllocWithId(numBytes, poolId, (void*)((uint32_t)__get_LR() | 0x80000000 ))
```

Description:

Defines an API to be used when allocating memory which will be used for long periods of time.

3.3.3 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.

Name:

```
typedef struct poolStat_tag
{
    uint16_t numBlocks;
    uint16_t allocatedBlocks;
    uint16_t allocatedBlocksPeak;
    uint16_t allocationFailures;
    uint16_t freeFailures;
#ifdef MEM_TRACKING
    uint16_t poolFragmentWaste;
    uint16_t poolFragmentWastePeak;
    uint16_t poolFragmentMinWaste;
    uint16_t poolFragmentMaxWaste;
#endif /*MEM_TRACKING*/
} poolStat_t;
```

Description:

Defines the statistical information which is recorded for every memory pool used for debugging.

Name:

```
typedef PACKED_STRUCT BlockTracking_tag
{
    void *blockAddr;
    uint16_t blockSize;
    uint16_t fragmentWaste;
    void *allocAddr;
    void *freeAddr;
    uint16_t allocCounter;
    uint16_t freeCounter;
```

```
    memTrackingStatus_t allocStatus;
    uint32_t timeStamp;
    void *pCaller;
}blockTracking_t;
```

Description:

Defines the memory buffer tracking information used for debugging.

Name:

```
typedef struct listHeader_tag
{
    listElement_t link;
    struct pools_tag *pParentPool;
}listHeader_t;
```

Description:

Defines the header of every memory buffer. This header is intended for the MemManager.

Name:

```
typedef struct pools_tag
{
    list_t anchor;
    uint16_t nextBlockSize;
    uint16_t blockSize;
    uint16_t poolId;
#ifdef MEM_STATISTICS
    poolStat_t poolStatistics;
#endif
    uint8_t numBlocks;
    uint8_t allocatedBlocks;
}pools_t;
```

Description:

Defines the structure of a memory pool.

Name:

```
typedef struct poolInfo_tag
{
    uint16_t blockSize;
    uint16_t poolSize;
    uint16_t poolId;
    uint8_t padding[2];
}poolInfo_t;
```

Description:

Defines the memory layout information.

3.3.4 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 and 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_BufferAllocWithId ()

Prototype:

```
void* MEM_BufferAllocWithId
(
    uint32_t numBytes,
    uint8_t poolId,
    void *pCaller
);
```

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 size.

Parameters:

Name	Type	Direction	Description
numBytes	uint32_t	[IN]	Requested buffer size, in bytes
poolId	uint8_t	[IN]	The Id of the pools where to search for a free memory buffer
pCaller	void*	[IN]	The address from where the function who called MEM_BufferAllocWithId() was called (for debug purpose)

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 a 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_BufferRealloc()

Prototype

```
void* MEM_BufferRealloc(void* buffer, uint32_t new_size);
```

Description

This function can be used to request a larger buffer.

MEM_BufferRealloc is based on standard C realloc() behavior:

- If new requested size is 0, acts like MEM_BufferFree and returns NULL.
- If input pointer is NULL, acts like MEM_BufferAllocate and returns pointer to allocated buffer.
- If the new requested size is smaller/equal to block size, returns input pointer. - If new requested size is bigger than block size, allocates a new buffer, copy the input buffer data to new buffer, frees the input buffer and returns pointer to new allocated buffer.
- If MEM_BufferAllocate fails, MEM_BufferRealloc returns a NULL pointer and doesn't free the input buffer

Parameters:

Name	Type	Direction	Description
buffer	void*	IN	Pointer to original buffer to be reallocated
new_size	uint32_t	IN	New size requested

Returns

Pointer to the newly allocated buffer or NULL if the allocation fails.

MEM_GetAvailableBlocks()

Prototype:

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

Description:

This function checks how many buffers are available with a size equal or greater than the specified size.

Parameters:

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

Returns:

The buffer count that satisfied the condition to have their size equal to or greater than the size 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_BufferCheck ()

Prototype:

```
uint8_t MEM_BufferCheck
(
    uint8_t *p,
    uint32_t size
);
```

Description:

The function is used internally by the FLib_MemCpy() and FLib_MemSet() functions to check for memory buffer overflow for debugging.

Parameters:

Name	Type	Direction	Description
p	uint8_t*	[IN]	Pointer to the location where data will be placed
size	uint32_t	[IN]	Number of bytes to write

Returns:

1 – Memory overflow detected

0 – No memory overflow

MEM_CheckIfMemBuffersAreFreed ()

Prototype:

```
void MEM_CheckIfMemBuffersAreFreed
(
    void
);
```

Description:

The function checks memory buffers that are allocated for more than the maximum duration specified for debugging.

This function may be called from the Idle Task.

Parameters:

None.

Returns:

None.

MEM_WriteReadTest()

Prototype:

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

Description:

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

This function is intended for module testing only.

Parameters:

None.

Returns:

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

MEM_GetTimeStamp()

Prototype:

```
extern uint32_t MEM_GetTimeStamp
(
    void
);
```

Description:

The function is used for debug purpose and should be implemented by the application.

Parameters:

None.

Returns:

This function returns a time-stamp value in milliseconds.

3.3.5 Sample code

```
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 ...
}
```

3.3.6 Memory manager debug support

The Memory Manager module incorporates a debug component which allows the possibility to debug, trace and create memory usage statistics.

To enable the debug support, one needs to define one or more of the following macros.

3.3.6.1 MEM_DEBUG_OUT_OF_MEMORY

When defined, it will halt program execution when the memory allocation fails because no suitable memory block was found. This macro requires the Panic module to be enabled to work.

One can use this macro to detect if a memory pool needs more buffers to be configured for the requested size.

3.3.6.2 MEM_DEBUG_INVALID_POINTERS

When defined, it will halt the program execution if it tries to free an invalid memory address. It also requires the Panic module to be enabled to work.

The conditions caught with this debug macro are:

- Memory address not in *memHeap[ ]*
- Corrupt memory buffer header or invalid pointer
- Multiple free attempts of the same memory
- Free of a memory buffer that is enqueued in a linked list.

3.3.6.3 MEM_TRACKING

Defining this macro will cause the Memory Manager to record the following information to be recorded for every memory buffer in the *memTrack[ ]* array:

- Memory block address – does no change after MEM_Init()
- Memory block size – does no change after MEM_Init()
- Fragment waste for the last allocation
- The program location of the last allocation of this memory block
- The program location of the last free of this memory block
- Count of the total number of allocation and free operations
- Buffer Status: allocated/free
- Timestamp of the last operation. The MEM_GetTimeStamp() function should be implemented for this to work.
- The caller address of the function who called the last allocate

Also, the Helper Function library module offers the possibility to debug situations where the memory copy/fill will overflow a memory buffer, corrupting the entire pool of memory by enabling the gFLib_CheckBufferOverflow_d macro along with the MEM_TRACKING.

3.3.6.4 MEM_DEBUG

Enabling this macro will cause the program execution to halt when an unexpected memory tracking condition occurs, or when the MEM_BufferCheck() function detects a memory block overflow.

This macro requires the Panic module to be enabled and the MEM_TRACK macro to be defined.

3.3.6.5 MEM_STATISTICS

The scope of this debug component is to allow efficient memory pools configuration. When enabled, the Memory Manager statistic component will record the following information for every memory pool:

- Total number of buffers – does no change after MEM_Init()
- Current and peak number of allocated buffers from a memory pool
- Number of allocation/free failures for this pool

When combined with the memory tracking component, the statistics will also include buffer fragment waste information:

- poolFragmentWaste – current memory waste for the entire pool. This value increases/decreases as the memory allocate/free operations are performed.
- poolFragmentWastePeak – peak memory waste for the entire memory pool
- poolFragmentMinWaste – minimum memory waste of a buffer from the pool

- poolFragmentMaxWaste – maximum memory waste of a buffer from the pool

Adjusting memory pools configuration

It is a good advice to over-estimate the initial memory pool configuration, and then enable the memory manager debug macros fine tune the number of buffers and their size at run time.

For example, if after the initial memory configuration, the system halts in panic because it ran out of memory, then the number of buffer for the memory pool appropriate for the requested size should be increased.

After the application runs correctly, the memory statistics can be used to further adjust the memory pools.

The *allocatedBlocksPeak* indicator can be analyzed for every pool and adjust the number of buffers so that: *numBlocks* – *allocatedBlocksPeak* <= 1. It is usually a good practice to leave a buffer for unpredicted situations.

The pool fragment waste indicators can be used to decrease the size of the buffers from a pool. The scope is to reduce the overall memory waste. Usually, the minimum waste should reach 0. The new size **must** be a multiple of 4 bytes.

3.3.6.6 DEBUG_ASSERT

This macro enables the MEM_ASSERT for the memory statistics component to catch unexpected conditions.

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
- Get the remaining time until a specified timer times out

Two types of timers are provided as follows:

- Single Shot Timers are run only once until they time out. They can be stopped before the timeout.
- Interval Timers are 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. Therefore, it must not call the blocking OS APIs. They can block but should never be used that way.

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 to reduce the loss of accuracy that may occur due to rounding errors in internal integer calculations.

The implementation of the Timers Manager on Kinetis MCU-based platforms uses either FTM or TPM peripheral. On the QN908X platforms, the RTC free-running counter is used. 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. For the Kinetis MCU-based platforms, this doesn't mean that the low-power timers run in low-power modes. Instead, 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. For the QN908X MCU-based platforms, the RTC timer is on during the low-power mode (except Power Down 1) and can generate interrupts.

The Timers' Manager creates a task to handle the internal processing. All callbacks are called in context and with the priority of the timer task. As a general guideline, callbacks must be non-blocking and short. They must not do more than issue a synchronization signal. The task is set up with an above-normal priority.

The Timers' Manager module also provides a timestamp functionality. This is implemented on top of the RTC for the Kinetis MCU-based platforms that can also use the PIT. In addition, it is possible to set the absolute time with a 30 milliseconds resolution and register an alarm event in absolute or relative time with a 1 second resolution. Note 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:

RTC is used as default hardware as a timestamp. If this define is set to TRUE, then the PIT HW is used as a timestamp.

Name:

```
#define gTMR_EnableLowPowerTimers_d     TRUE
```

Description:

Enables/disables the timer synchronization after exiting 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     4
```

Description:

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

Name:

```
#define gTmrStackTimers_c     4
```

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     600
```

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 an 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 )
#define gTmrAllTypes_c                           ( gTmrSingleShotTimer_c | gTmrIntervalTimer_c | \
                                                   gTmrSetMinuteTimer_c | gTmrSetSecondTimer_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. 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, the timer 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:

```
tmrErrCode_t 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:

The error code.

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 stops the timer and reconfigures it with 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:

```
tmrErrCode_t 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:

The error code.

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:

The remaining time in milliseconds until the timer expires. The function also returns 0 if the tmrId is invalid.

TMR_GetFirstExpireTime

Prototype:

```
uint32_t TMR_GetFirstExpireTime
(
    tmrTimerType_t timerType
);
```

Description:

The function returns the time in milliseconds until the first timer of the specified type expires (times out) or 0xFFFFFFFF if the specified timer type is not found.

Parameters:

Name	Type	Direction	Description
timerType	tmrTimerType_t	[IN]	Bitmask of timer types

Returns:

The remaining time in milliseconds.

TMR_GetFirstWithParam

Prototype:

```
uint32_t TMR_GetFirstWithParam
(
    void * param
);
```

Description:

Returns the ID of the first timer that has the specified parameter.

Parameters:

Name	Type	Direction	Description
Param	void *	[IN]	Parameter to compare

Returns:

Timer ID

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 time,
    pfTmrCallBack_t callback,
    void *param
);
```

Description:

Starts a low-power timer. When the timer goes OFF, call the callback function in a non-interrupt context. If the timer is running when this function is called, it is 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

Table continues on the next page...

Table continued from the previous page...

Name	Type	Direction	Description
timerType	tmrTimerType_t	[IN]	The type of the timer
time	uint32_t	[IN]	Time in ticks
callback	pfTmrCallBack_t	[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,
    pfTmrCallBack_t callback,
    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
callback	pfTmrCallBack_t	[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,
    pfTmrCallBack_t callback,
```

```
    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
callback	pfTmrCallBack_t	[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,
    pfTmrCallBack_t callback,
    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
callback	pfTmrCallBack_t	[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,
    pfTmrCallBack_t callback,
    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
callback	pfTmrCallBack_t	[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 ms).

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

Table continues on the next page...

Table continued from the previous page...

Name	Type	Direction	Description
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 and Non-Volatile Storage Module

3.5.1 Overview

In a standard Harvard-architecture-based MCU, the flash memory is used to store the 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 non-volatile data. The flash memories have individually erasable segments (sectors) and each segment has a limited number of erase cycles. If the same segments are used to store various kinds of data all the time, those segments quickly become unreliable. 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 subsequent sections. The program and erase memory operations are handled by the *Nvidle*() function. Before resetting the MCU, *NvShutdown* must be called to ensure that all saves are processed.

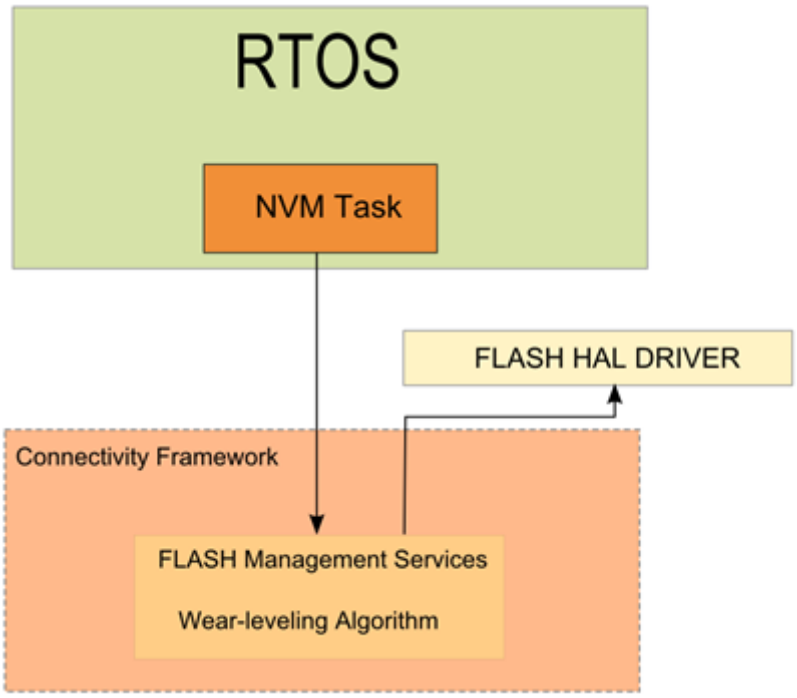


Figure 2. NVM overview

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 NV system uses for permanent storage and its boundaries are defined in the linker configuration file. The reserved

memory consists of two 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 and Non-Volatile Storage Module holds a pointer to a RAM table, where the upper layers register information about 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 contains, the size of a single element, a table entry ID, and an entry type.

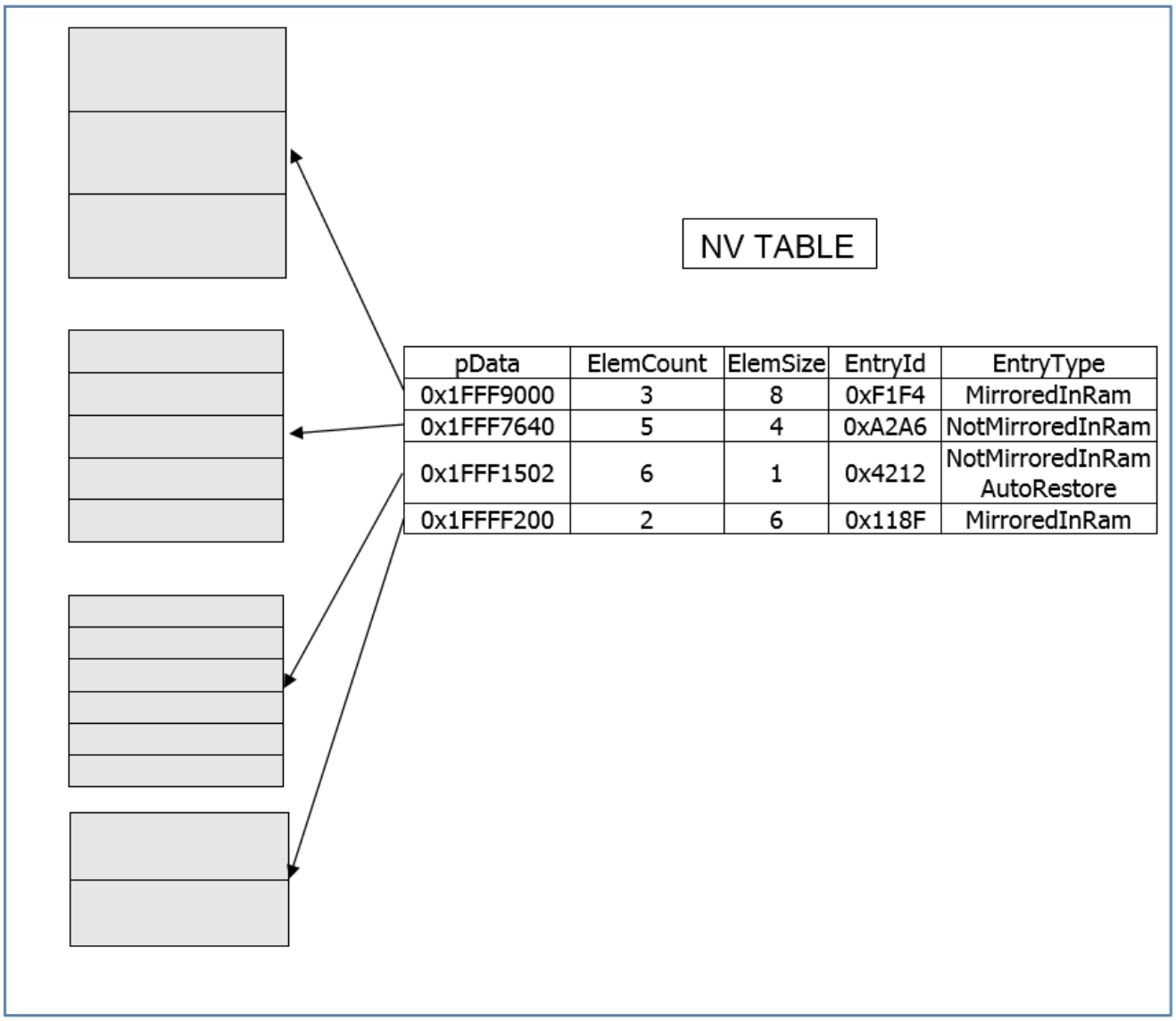
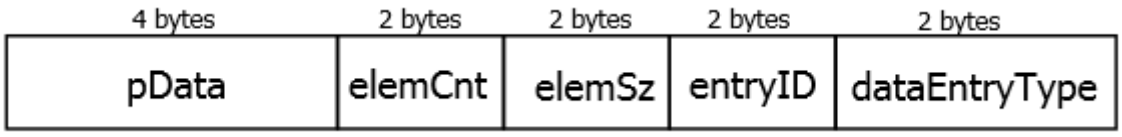


Figure 3. NV Table example

A RAM table entry has the following structure:



- **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/unmirrored auto restore)

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

Mirrored datasets require the data to be permanently kept in RAM, while unmirrored datasets have dataset entries either in flash or in RAM. If the unmirrored entries must be restored at the initialization, NotMirroredInRamAutoRestore should be used. 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.

Table 2. Mirrored data entry example

pData	0x1FFF8000
elemCnt	4
elemSz	10
entryID	1
dataEntryType	gNVM_MirroredInRam_c

Table 3. pData

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

Figure 4. The structure of a NV table entry

Table 4. Unmirrored data entry example

pData	0x1FFF8000
elemCnt	4
elemSz	10
entryID	1
dataEntryType	gNVM_NotMirroredInRam_c/ gNVM_NotMirroredInRamAutoRestore_c

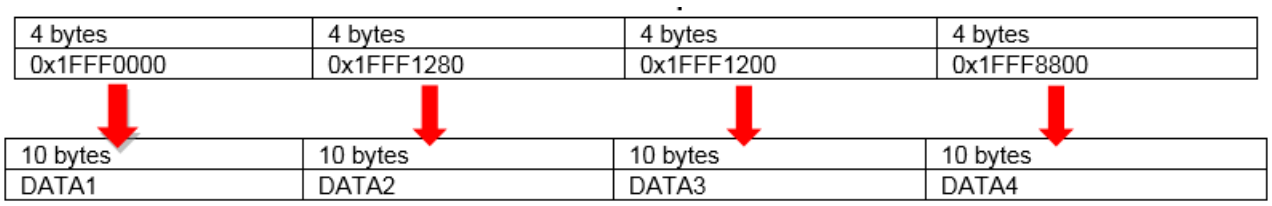


Figure 5. pData

When the data pointed to by the table entry pointer (pData) has changed (entirely or just a single element), the upper layers call the appropriate API function that requests the storage system to save the modified data. All the save operations (except for the synchronous save and atomic save) and 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. However, the application may choose another priority.

The saves are done in one virtual page, which is the active page. After a save is performed on an unmirrored dataset, pData points to a flash location and the RAM pointer is freed. As a result, the effective data should always be allocated using the memory management module.

The active page contains information about the records and the records. The storage system can save individual elements of a table entry or the entire table entry. Unmirrored datasets can only have individual saves.

On mirrored datasets, the save/restore functions must receive the pointer to 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 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, whether it is a single element or an 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 uninitialized table entries, declared explicitly to register new table entries at runtime. A new table entry can also replace an existing one if the register table entry is called with an 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 shown below:

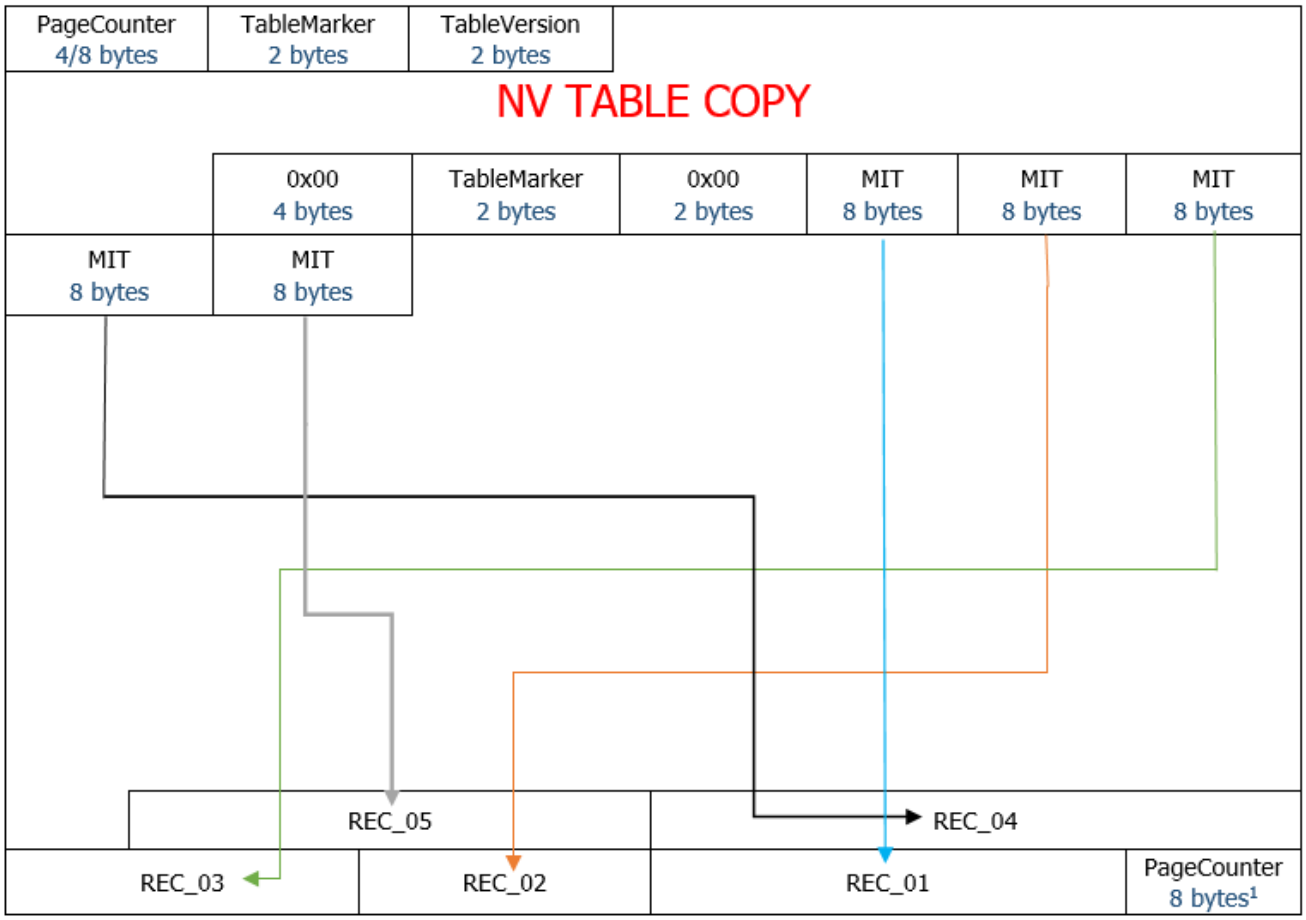


Figure 6. The layout of a flash active page

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 RAM are not copied. A flash copy of a RAM table entry has the following structure:

Figure 7. 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/unmirrored auto restore) • 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.

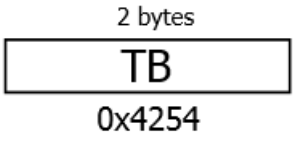


Figure 8. Table Marker

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:

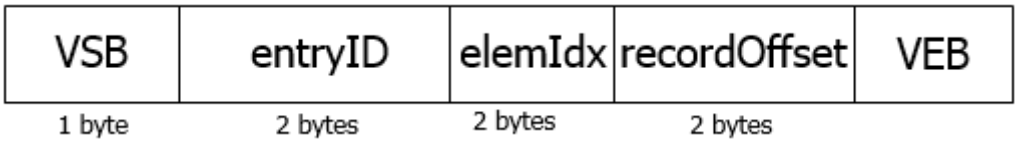


Figure 9. Meta Information Tag

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

Because the records are written to the flash page, the available page space decreases. As a result, the page becomes full and 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 this case, the latest saved datasets (table entries) are copied to virtual page 2.

In the following example, there are five datasets (one color for each dataset) with 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

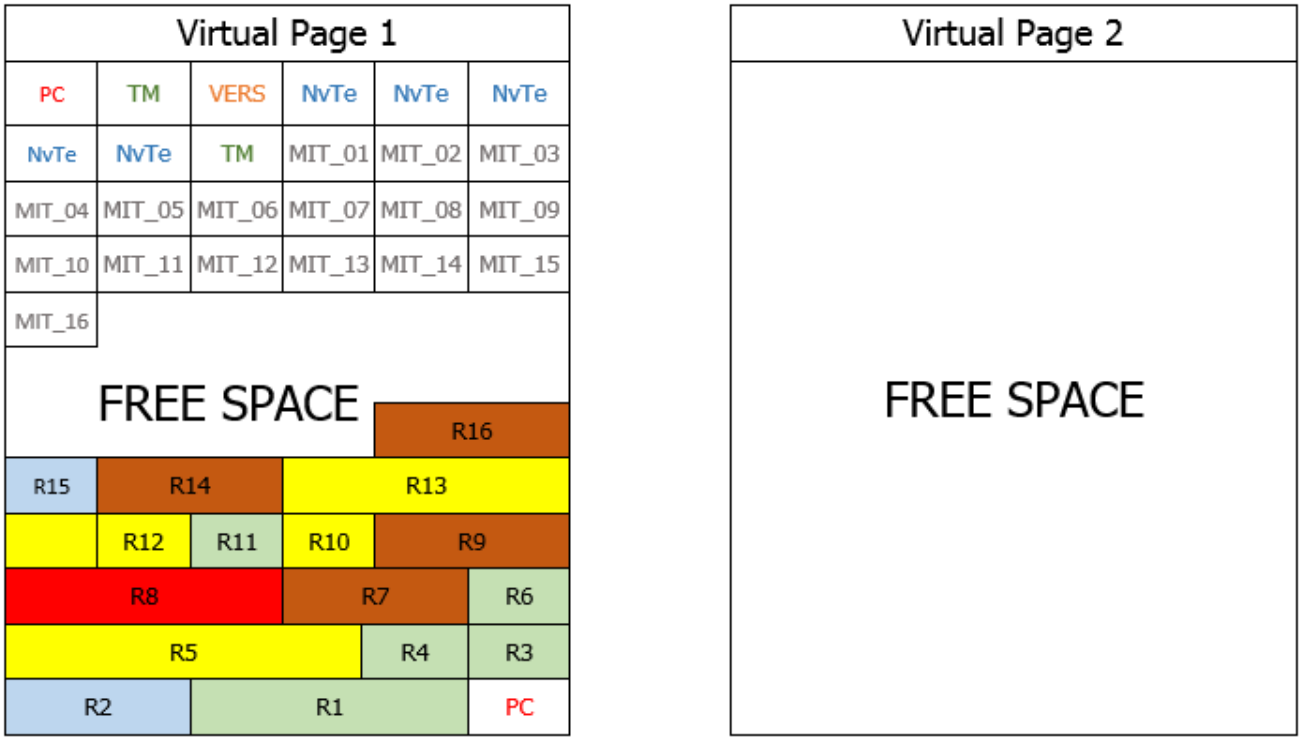


Figure 10. Virtual page 1 free space is not large enough for saving a new dataset

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.

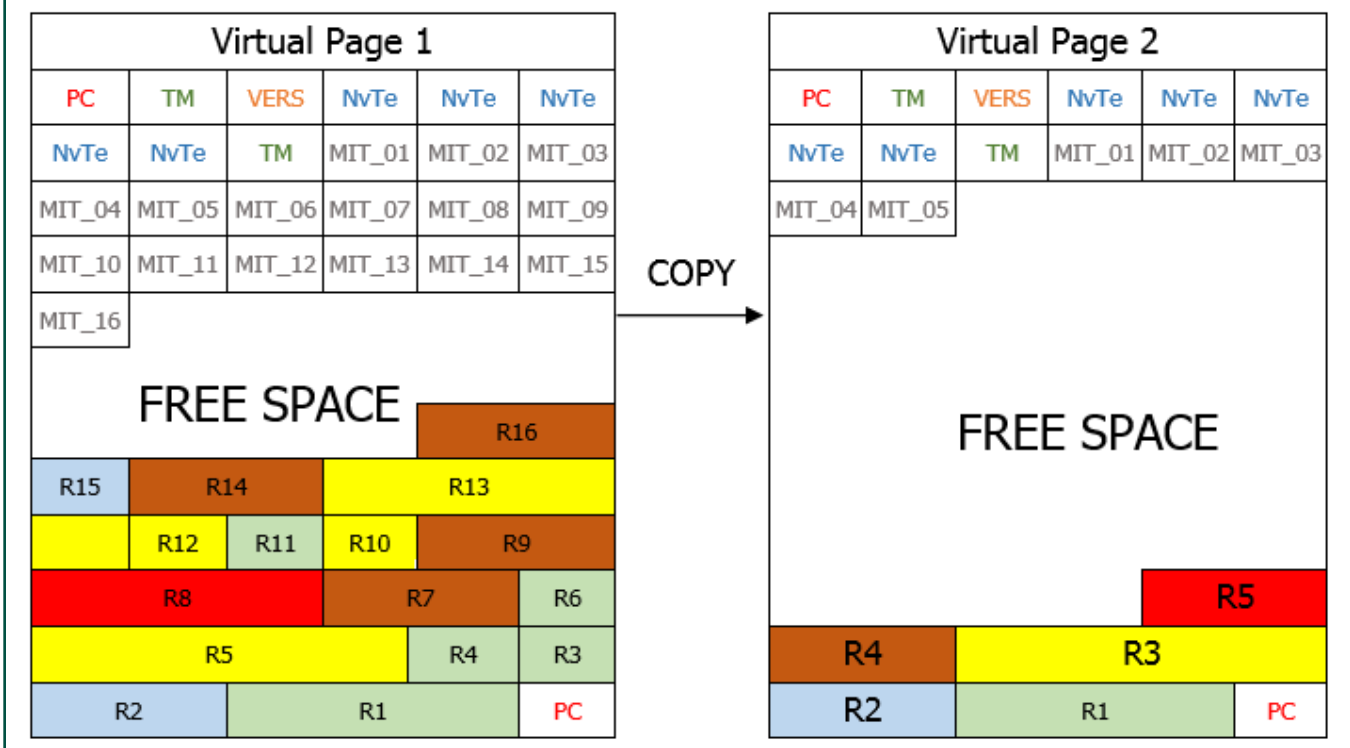


Figure 11. Page Copy

Finally, the virtual page 2 is validated by writing the PC value and a request to erase virtual page 1 is performed. The page is erased on an idle task, sector by sector where 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 must allocate the pData and change the RAM entry. It can choose to ignore the flash entry if the entry is not desired. If any entry from RAM differs from its flash equivalent at initialization, a page copy is triggered that ignores the entries that are different. In other words, data stored in those entries will be lost.

The application can check if the RAM table was updated, in other words, if 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 detects the update and automatically upgrades the flash table. The upgrade process triggers a page copy that moves the flash data from the active page to the other one. It keeps the entries that were not modified intact and it moves the entries that had their elements count changed as follows:

- If the RAM element count is smaller than the flash element count, the upgrade only copies as much elements as are in RAM
- If the RAM element count is larger than the flash element count, the upgrade copies all data from flash and fills the remaining space with data from RAM

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

NOTE

This section is not applicable to JN5189/QN9090/K32W061.

The FlexNVM can be partitioned to support a simulated EEPROM subsystem. All the read/write operations are done in the FlexRAM area. Every time 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 the 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 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 the FlexRAM start address. After all datasets 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 eight NV table entries (datasets) are 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).

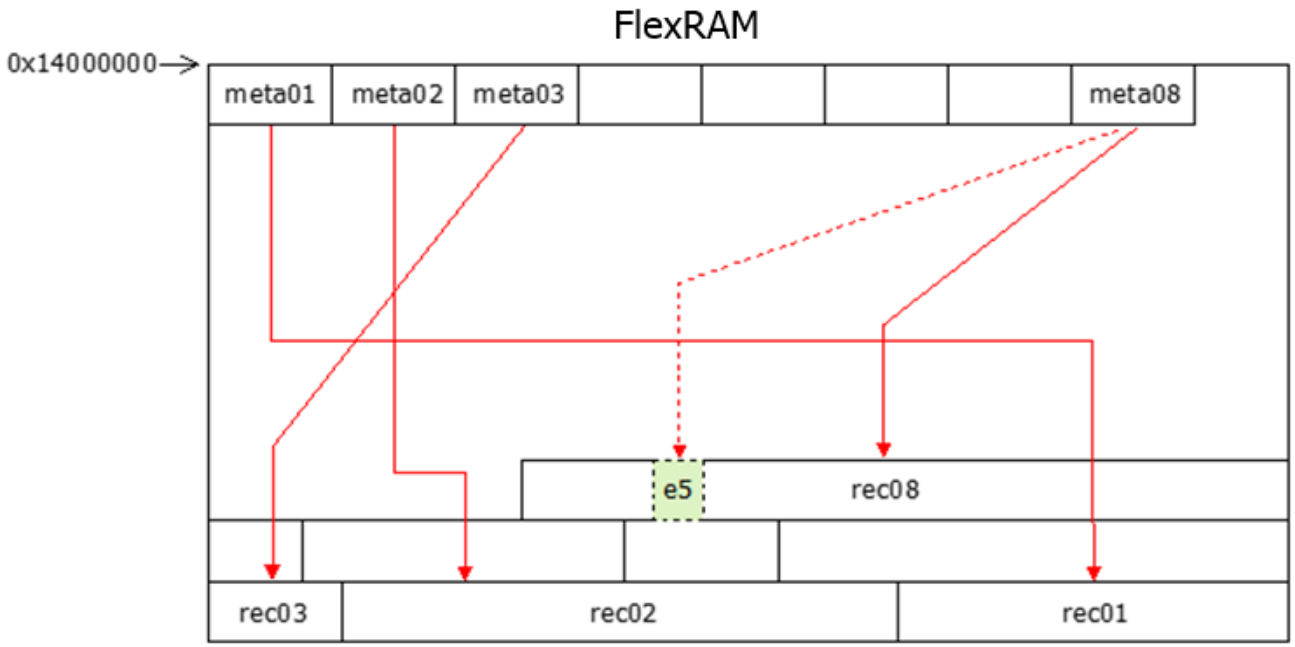


Figure 12. FlexRAM memory used to store NV table entries

If Flex NVM is enabled, unmirrored datasets are not supported. Neither the extended functionality is supported.

3.5.4 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 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 8. 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. The 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 enable overwriting older entries in the pending saves queue (if it is full). If it is FALSE and the queue is full, the module tries to process the oldest save in the queue to free a position. It is set to FALSE by default.

Name:

```
gNvMinimumFreeBytesCountStart_c
```

Description:

Macro used to define the minimum free space at initialization. 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 NVM table was updated. It is set to 1 by default. The application should modify this every time the NVM table is updated and the data from NVM is still required.

Name:

```
gNvTableKeptInRam_d
```

Description:

Set gNvTableKeptInRam_d to FALSE, if the NVM table is stored in FLASH memory (default). If the NVM table is stored in RAM memory, set the macro to TRUE.

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_SaveRequestRejected_c,
    gNVM_InvalidTimerID_c,
    gNVM_MissingEndOfTableMarker_c,
    gNVM_NvTableExceedFlexRAMSize_c,
    gNVM_NvWrongFlashDataIFRMap_c,
    gNVM_CannotCreateMutex_c,
    gNVM_NoMemory_c,
    gNVM_IsMirroredDataSet_c,
    gNVM_DefragBufferTooSmall_c,
    gNVM_FragmentatedEntry_c
} NVM_Status_t;
```

Description:

Enumerated data type definition for NV storage module error codes.

3.5.6 Flash Management and Non-Volatile Storage Module API primitives

NvModuleInit

Prototype:

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

Description:

This function is used to initialize the Flash Management and Non-Volatile Storage 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 is triggered and the different entries are skipped in the process. It also handles NVM table changes. For example, the MCU program was changed and the NVM table was updated by automatically 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 NvIdle() function is called. If ptrData belongs to an unmirrored dataset, after the save the RAM pointer is freed and pData points 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 NvIdle() function is called. If ptrData belongs to an unmirrored dataset, after the save the RAM pointer is freed and pData points 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 is made. Otherwise, only the element indicated by the ptrData is saved. If the function is called before a previous save on interval was processed, the call is 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. When that counter reaches zero, the dataset is saved as soon as the NvIdle() function is called. If the ptrData belongs to an unmirrored dataset, after the save, the RAM pointer is freed and pData points 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 is made. Otherwise, only the element indicated by ptrData is saved.

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
```

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
);
```

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

Returns:

One of the following:

```
gNVM_OK_c
gNVM_ModuleNotInitialized_c
gNVM_CriticalSectionActive_c
gNVM_PointerOutOfRange_c
gNVM_NullPointer_c
gNVM_SaveRequestRejected_c
```

NvAtomicSave

Prototype:

```
NVM_Status_t NvAtomicSave
(
    void
);
```

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:

None.

Returns:

One of the following:

```
gNVM_OK_c
gNVM_ModuleNotInitialized_c
gNVM_CriticalSectionActive_c
gNVM_PointerOutOfRange_c
gNVM_NullPointer_c
gNVM_SaveRequestRejected_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 automatically 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 only restores a pointer to the flash location of the entry. In order to restore the data from flash to RAM, the application has 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 sync 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. This function guarantees that all the sync save/erase/copy operations are put in the pending queue while the critical section is on and processed when the critical section is lifter, on the idle task.

Parameters:

None.

Returns:

None.

NvClearCriticalSection

Prototype:

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

Description:

This function decrements an internal counter variable each time it is called. All the sync 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. Any saves that were not processed when the critical section was active will be processed here. It must be called from a low priority task, such as Idle 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 registering a new table entry or updating 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. If the critical section is active, the page copy is done as soon as it is deactivated. 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

Table continues on the next page...

Table continued from the previous page...

Name	Type	Direction	Description
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 is triggered and the data associated with the entry is not copied. If the critical section is active, the page copy is done as soon as it is deactivated. It sets entrySize and entryCount to 0 and pData to NULL. EntryId remains unchanged. For unmirrored datasets, use NvErase because it can delete individual elements. The 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. The 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 initialization if some flash entries are different than the RAM counterparts. To use data stored in the flash table, the application should copy the members returned by this function in their respective RAM entry and allocate pData. The 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
gNVM_AddressOutOfRange_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 NVM table to point to the new location. If the specified element is already in RAM, the function cancels any pending saves and returns. The 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 function erases an unmirrored dataset from flash. It generally does not trigger a page copy, but writes a new record in the NVM memory that specifies that this element was removed. If the page is full, a page copy is triggered. It sets pData for the specified element to NULL. It can only erase a single element. If the specified element is in RAM, it is freed but no changes are made in flash. The 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
```

NvShutdown

Prototype:

```
void NvShutdown
(
    void
);
```

Description:

This function blocks until all the saves in queue, page copy operations, and interval saves have been processed to ensure that the MCU has the latest data before a reset.

Parameters:

None.

Returns:

None.

CompletePendingOperations

Prototype:

```
void NvCompletePendingOperations
(
    void
);
```

Description:

This function attempts to complete all the NVM related pending operations, like queued writes, page copy, page erase and save on interval requests.

Parameters:

None

Returns:

None

3.5.7 Production Data Storage

NOTE

The Production Data Storage is not used for the QN908X MCU-based platforms. NVDS is used instead.

3.5.7.1 Overview

Different platforms/boards need board/network node-specific settings (in other words, 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, for example, a silicon mounted on a PCB in a specific configuration, rather than to just the silicon itself.

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

NOTE

This sector is not erased/written at code download time and it is not updated via over-the-air firmware update procedures 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 flash, the firmware does not take into account the respective values corresponding to node-specific data.

3.5.7.2 Constant Definitions

Name:

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

Description:

This symbol is defined in the linker file. It specifies the start address of the *FREESCALE_PROD_DATA* section.

Name:

```
static const uint8_t mProdDataIdentifier[10] = {"PROD_DATA:"};
```

Description:

The value of this constant is copied as identification word (header) at the beginning of the *FREESCALE_PROD_DATA* area and verified by the dedicated read function.

3.5.7.3 Data type definitions

Name:

```
typedef PACKED_STRUCT hardwareParameters_tag{
    uint8_t identificationWord[10];
    uint8_t reserved[32];
    uint8_t ieee_802_15_4_address[8];
    uint8_t bluetooth_address[6];
    uint32_t xtalTrim;
    uint32_t edCalibrationOffset;
    uint32_t pllFstepOffset;
    uint32_t gInternalStorageAddr;
    uint16_t hardwareParamsCrc;
}hardwareParameters_t;
```

Description:

Defines the structure of the hardware-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 and the backwards compatibility must be maintained.

The CRC calculation starts from the *reserved* field of the *hardwareParameters_t* and ends before the *hardwareParamsCrc* field. Additional members to this structure may be added using the following methods:

- Add new fields between *gInternalStorageAddr* and *hardwareParamsCrc* fields. This will cause a CRC fail at the next parameter read through the dedicated function due to additional bytes fed to the CRC calculator. The user must fill a

hardwareParameters_t variable with the previous values (raw copy from flash) and rewrite them to the flash memory using the dedicated function.

- Add new fields before the *gInternalStorageAddr* field. This method will not cause a CRC fail, but the user must keep in mind to subtract the total size of the new fields from the size of the reserved field. For example, if a field of *uint8_t size* is added using this method, the size of the reserved field shall be changed to 31.

3.5.7.4 API Primitives

NV_ReadHWParameters

Prototype:

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

Description:

Verifies that the dedicated flash area starts with the identification information and that data is valid using a CRC. If any of the checks fail, it fills the RAM data structure with *0xFF*, otherwise it loads the hardware-dependent information into it.

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:

Stores the hardware-dependent information into flash. Copies the identification information, calculates and updates the CRC value. It is recommended to use a read-modify-write sequence when hardware-specific data needs to be updated.

Parameters:

Name	Type	Direction	Description
pHwParams	hardwareParameters _t*	[IN]	Pointer to a data structure containing hardware-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 used for random number generation. It uses hardware RNG peripherals and a software pseudo-random number generation algorithm. If no hardware acceleration is present, the RNG module uses a software algorithm. On devices that have the SIM_UID registers, the UIDL is used as the initial seed for the random number generator.

3.6.2 Constant macro definitions

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: Generates a random number using a polynomial.

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: The 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: This function stops 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, USART, USB, SPI, I2C), using the same API.

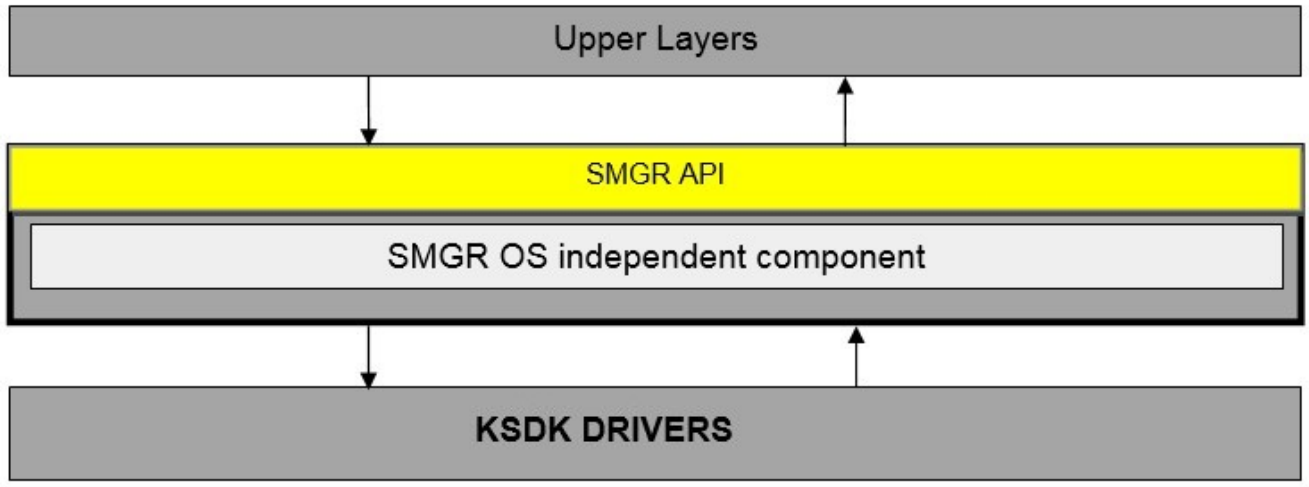


Figure 13. Serial manager overview

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 is 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. 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). 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
#define gSerialMgrUseCustomInterface_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 I2C.

Name:

```
#define gSerialMgrRxBufSize_c 32
```

Description:

Defines the RX buffer size.

Name:

```
#define gSerialMgrTxQueueSize_c 5
```

Description:

Defines the TX queue size.

Name:

```
#define gSerialTaskStackSize_c 1024
```

Description:

Defines the serial manager task stack size.

Name:

```
#define gSerialTaskPriority_c 3
```

Description:

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

Name:

```
#define gSerialMgrUseFSCIHdr_c 0
```

Description:

Defines usage of FSCI header and packet separation in SPI transfers for an SPI Master device.

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:

To print a hexadecimal number, choose between BigEndian=1/LittleEndian=0, newline, commas, or spaces (between bytes).

3.9.3 Data type definitions

Name:

```
typedef enum
{
    gSerialMgrNone_c      = 0,
    gSerialMgrUart_c      = 1,
    gSerialMgrUSB_c       = 2,
    gSerialMgrUSB_VNIC_c  = 3,
    gSerialMgrIICMaster_c = 4,
    gSerialMgrIICSlave_c  = 5,
    gSerialMgrSPIMaster_c = 6,
    gSerialMgrSPISlave_c  = 7,
    gSerialMgrLpuart_c    = 8,
    gSerialMgrLpsci_c     = 9,
    gSerialMgrCustom_c    = 10,
    gSerialMgrUsart_c     = 11,
}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* param);
```

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              = 0,
   gSerial_InvalidParameter_c     = 1,
   gSerial_InvalidInterface_c     = 2,
   gSerial_MaxInterfacesReached_c = 3,
   gSerial_InterfaceNotReady_c    = 4,
   gSerial_InterfaceInUse_c       = 5,
   gSerial_InternalError_c        = 6,
   gSerial_SemCreateError_c       = 7,
   gSerial_OutOfMemory_c          = 8,
   gSerial_OsError_c              = 9,
}serialStatus_t;
```

Description:

Serial manager status codes.

3.9.4 API primitives

SerialManager_Init ()

Prototype:

```
void Serial_InitManager( void );
```

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

Parameters:

None.

Returns:

None.s

Serial_InitInterface ()

Prototype:

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

Description: Initializes a communication interface.

Parameters:

Name	Type	Direction	Description
InterfaceId	uint8_t *	[IN]	Interface ID
interfaceType	serialInterfaceType_t	[IN]	The type of interface
instance	uint8_t	[IN]	The instance of the HW module (ex: if UART1 is used, this value should be 1)

Returns:

- gSerial_InterfaceInUse_c if the interface is already opened
- gSerial_InvalidInterface_c if the interface is invalid
- gSerial_SemCreateError_c if the 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 is received.

Parameters:

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

Returns:

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

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.

Serial_CustomReceiveData ()

Prototype:

```
uint32_t Serial_CustomReceiveData(uint8_t InterfaceId, uint8_t *pRxData, uint32_t size);
```

Description: This function must be called by a custom interface to notify the Serial Manager that a specified number of bytes have been received.

Parameters:

Name	Type	Direction	Description
InterfaceID	uint8_t	[IN]	Interface ID
pRxData	uint8_t*	[IN]	Pointer to the received data
Size	uint32_t	[IN]	Number of bytes received

Returns:

The number of bytes that could not be stored by Serial Manager.

Serial_CustomSendData ()

Prototype:

```
uint32_t Serial_CustomSendData (uint8_t *pData, uint32_t size);
```

Description: This function must be implemented by the custom interface, and it is called by the Serial Manager to transmit data over the custom interface.

Parameters:

Name	Type	Direction	Description
pData	uint8_t*	[IN]	Pointer to the data
Size	uint32_t	[IN]	Number of bytes to transmit

Returns:

Error code, or 0 if success.

Serial_CustomSendCompleted ()

Prototype:

```
void Serial_CustomSendCompleted (uint8_t InterfaceId);
```

Description: This function must be called by a custom interface to notify the Serial Manager that the current data transmission has ended.

Parameters:

Name	Type	Direction	Description
InterfaceId	uint8_t	[IN]	Interface ID

Returns:

None.

3.10 ModuleInfo

3.10.1 Overview

The ModuleInfo is a small Connectivity Framework module that provides a mechanism that allows stack components to register information about themselves.

The information comprises:

- Component or module name (for example: Bootloader, IEEE 802.15.4 MAC, and Bluetooth LE Host) and associated version string

- Component or module ID

- Version number

- Build number

The information can be retrieved using shell commands or FSCI commands .

3.10.2 User-defined data type definitions

Name:

```
typedef PACKED_STRUCT moduleInfo_tag {
  const char **  moduleString;
```

```
  uint8_t  moduleId;
  uint8_t  versionNumber[3];
  uint16_t buildNumber;
  uint8_t  padding[2];
} moduleInfo_t;
```

Description:

Module information data structure.

Field description:

- moduleString - the module information string that is usually built using concatenation macros between module name and module version.
- moduleId - a hexadecimal value that can be used as an unique ID within the system
- versionNumber - three decimal values that specifies the major, minor and patch version of the component/module
- build number - the build number

3.10.3 ModuleInfo API primitives

RegisterModuleInfo

Prototype:

```
RegisterModuleInfo(moduleName, moduleNameString, moduleId, versionNoMajor, versionNoMinor,
versionNoPatch, buildNo)
```

Description: Registers a new component or module information. This is not actually a function but a macro that will create in FLASH memory a structure that will keep the component version information in a dedicated section called VERSION_TAGS.

Parameters:

Name	Type	Direction	Description
moduleName	char[]	[IN]	Module name
moduleNameString	const char**	[IN]	Module name as a concatenated string of name and version
moduleId	uint8_t	[IN]	Module ID
versionNoMajor	uint8_t	[IN]	Major version number
versionNoMinor	uint8_t	[IN]	Minor version number
versionNoPatch	uint8_t	[IN]	Patch version number
buildNo	uint16_t	[IN]	Build number

Returns:

None.

3.11 Framework Serial Communication Interface

3.11.1 Overview

The Framework Serial Communication Interface (FSCI) is both a software module and a protocol that allows monitoring and extensive testing of the protocol layers. 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 Test Tool software is an example of a host processor, which can interact with FSCI Black Boxes at various layers. In this setup, the user can run numerous commands to test the Black Box application services and interfaces.

The FSCI enables common service features for each device 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 triggers 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, Test Tool can communicate with each MAC layer over two UART interfaces.

NOTE

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

3.11.2 FSCI packet structure

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

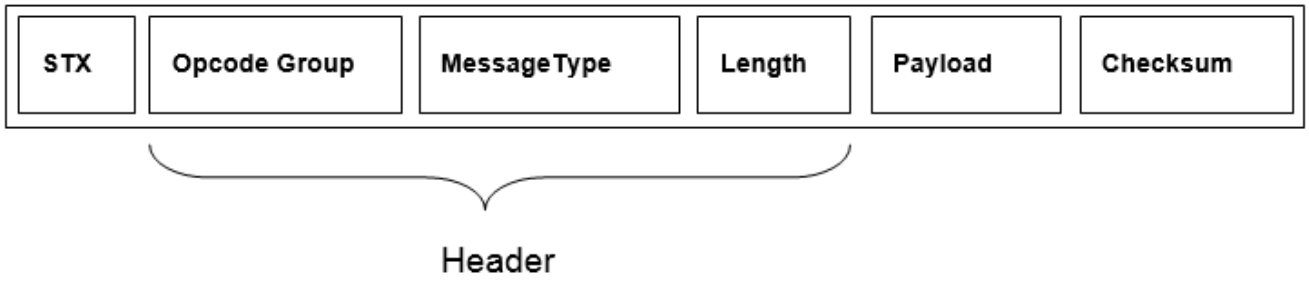


Figure 14. Packet structure

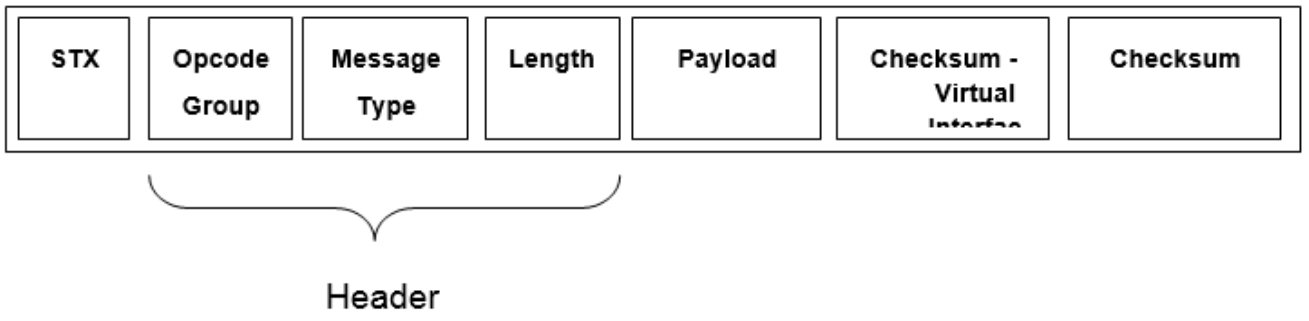


Figure 15. Packet structure when virtual interfaces are used

Table 5. 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 or 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 or 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. The second checksum is used for error detection.

3.11.3 Constant macro definitions

Name:

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


#define gFsciMaxOpGroups_c 8
#define gFsciMaxInterfaces_c 1
#define gFsciMaxVirtualInterfaces_c 0
#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) */
#define gFsciHostMacSupport_c 0 /* Host support at MAC layer */
#define gFsciHostSyncUseEvent_c 0 /* Use event for receiving a sync response */
#define gFsciTxAck_c 0 /* Tx ACK packet for each received packet */
#define gFsciRxAck_c 0 /* Expect ACK after each sent packet */
#define gFsciRxTimeout_c 0 /* boolean - FSCI Restarts start marker search if no character is received
during mFsciRxRestartTimeoutMs_c milliseconds */
```

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, application, and FSCI.

3.11.4 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 */

    mFsciOtaSupportSetModeReq_c = 0x28,
    mFsciOtaSupportStartImageReq_c = 0x29,
    mFsciOtaSupportPushImageChunkReq_c = 0x2A,
    mFsciOtaSupportCommitImageReq_c = 0x2B,
    mFsciOtaSupportCancelImageReq_c = 0x2C,
    mFsciOtaSupportSetFileVerPoliciesReq_c = 0x2D,
```

```
    mFsciOtaSupportAbortOTAUpgradeReq_c = 0x2E,
    mFsciOtaSupportImageChunkReq_c = 0x2F,
    mFsciOtaSupportQueryImageReq_c = 0xC2,
    mFsciOtaSupportQueryImageRsp_c = 0xC3,
    mFsciOtaSupportImageNotifyReq_c = 0xC4,

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

    mFsciMsgAllowDeviceToSleepReq_c = 0x70, /* Fsci-SelectWakeUpPIN.Request */
    mFsciMsgWakeUpIndication_c = 0x71, /* Fsci-WakeUp.Indication */
    mFsciMsgReadExtendedAdrReq_c = 0xD2, /* Fsci-ReadExtAddr.Request */
    mFsciMsgGetWakeUpReasonReq_c = 0x72, /* Fsci-GetWakeUpReason.Request */
    mFsciMsgWriteExtendedAdrReq_c = 0xDB, /* Fsci-WriteExtAddr.Request */

    mFsciMsgError_c = 0xFE, /* FSCI error message. */
    mFsciMsgAck_c = 0xFD, /* FSCI acknowledgment.*/
    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 fsciInterface);
```

Description:

Defines the message handler function type.

Name:

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

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 the 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 stored at payload[len]. The checksum field insures that there is always space for it, even when 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.

3.11.5 FSCI API primitives

FSCI_Init ()

Prototype:

```
void FSCI_Init(gFsciSerialConfig_t* pConfig);
```

Description: Initializes the FSCI internal variables.

Parameters:

Name	Type	Direction	Description
initStruct	gFsciSerialConfig_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 pfHandler,
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
pfHandler	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

Table continues on the next page...

Table continued from the previous page...

Name	Type	Direction	Description
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_LogFormatedText

Prototype:

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

Description: Sends a formatted text string to the host.

Parameters:

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

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_transmitFormatedPacket

Prototype:

```
void FSCI_transmitFormatedPacket( void clientPacket_t *pPkt, uint32_t fsciInterface );
```

Description: Sends 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 msgLen, uint32_t interfaceId);
```

Description: Encodes and sends 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
msgLen	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. 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.

FSCI_Ack

Prototype:

```
void FSCI_Ack( uint8_t checksum, uint32_t fsciInterface );
```

Description:

Sends an ACK packet over the serial interface with an optional checksum identifier. This function does not use dynamic memory. The packet is sent in blocking mode. It is recommended to enable the ACK functionality through the gFsciTxAck_c macro definition.

Parameters:

Name	Type	Direction	Description
checksum	uint8_t	[IN]	Packet identifier to be included in the ACK, currently optional.
fsciInterface	uint32_t	[IN]	The interface on which the packet must be sent

Returns:

None.

3.11.6 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 a Black Box on a board, and MAC higher layers are 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. 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 the 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 polls 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, available for RTOS environments, is using 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 the 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 because the blocked task is the Serial Manager task, which reaches a deadlock as cannot be released again.

3.11.7 FSCI ACK

The FSCI provides an FSCI packet reception validation and retransmissions through an ACK mechanism. The ACK submodule has two components than can be enabled independent of one another, ACK transmission and ACK reception.

ACK transmission is enabled through the gFsciTxAck_c macro definition. Each FSCI valid packet received triggers an FSCI ACK packet transmission on the same FSCI interface that the packet was received on. The serial write call is performed synchronously to send the ACK packet before any other FSCI packet and only then the registered handler is called to process the received packet.

The ACK is represented by the gFSCI_CnfOpcodeGroup_c and mFsciMsgAck_c Opcode. An additional byte is left empty in the payload that can be used optionally as a packet identifier to correlate packets and ACKs.

ACK reception is the other component that is enabled through gFsciRxAck_c. The behavior such that every FSCI packet sent through a serial interface triggers an FSCI ACK packet reception on the same interface after the packet is sent. If an ACK packet is received, the transmission is considered successful. Otherwise, the packet is resent a number of times.

The ACK wait period is configurable through mFsciRxAckTimeoutMs_c and the number of transmission retries through mFsciTxRetryCnt_c.

The above ACK mechanism can also be coupled with a FSCI packet reception timeout enabled through gFsciRxTimeout_c and configurable through mFsciRxRestartTimeoutMs_c. Whenever there are no more bytes to be read from a serial interface, a timeout is configured at the predefined value should no other bytes be received. If new bytes are received, the timer is stopped and eventually canceled at successful reception. However, if for any reason the timeout is triggered, the FSCI module considers that the current packet is invalid, drops it and searches for a new start marker.

3.11.8 FSCI usage example

Initialization

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

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

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

Registering operation groups

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

```
…
FSCI_RegisterOpGroup( myOpGroup, gFsciMonitorMode_c, myHandlerFunc, myParam,
myInterface );
```

Implementing handler function

```
void fsciMcpsReqHandler(void *pData, void* param, uint32_t interfaceId)
{
    clientPacket_t *pClientPacket = ((clientPacket_t*)pData);
    fsciLen_t myNewLen;

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

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

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

    default:
        MEM_BufferFree( pData );
        FSCI_Error( gFsciUnknownOpcode_c, interfaceId );
        return;
    }
    /* Free message received over the serial interface */
    MEM_BufferFree( pData );
}
```

3.12 Security Library

3.12.1 Overview

The framework provides support for cryptography in the security module. It supports both software and hardware encryption. Depending on the device, the hardware encryption uses either the MMCAU, LTC or CAU3 module instruction set or dedicated AES and SHA hardware blocks. Using the hardware support directly requires the input data to be four bytes aligned.

Both implementations are supplied in a library format.

3.12.2 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.

3.12.3 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

Table continues on the next page...

Table continued from the previous page...

Name	Type	Direction	Description
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

Table continues on the next page...

Table continued from the previous page...

Name	Type	Direction	Description
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:

```
uint8_t AES_128_CCM(uint8_t* pInput,
    uint16_t inputLen,
    uint8_t* pAuthData,
    uint16_t authDataLen,
    uint8_t* pNonce,
    uint8_t nonceSize,
    uint8_t* pKey,
    uint8_t* pOutput,
    uint8_t* pCbcMac,
    uint8_t macSize,
    uint32_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	uint16_t	[IN]	Input message length in bytes.
pAuthData	uint8_t*	[IN]	Pointer to the additional authentication data
authDataLen	Uint16_t	[IN]	The length of the additional authentication data.
pNonce	uint8_t*	[IN]	Pointer to the Nonce
nonceSize	uint8_t	[IN]	The length of the Nonce 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.
pCbcMac	uint8_t*	[IN/OUT]	Encryption: pointer to the location to store the authentication code. Decryption: pointer to the location of the received authentication code
macSize	uint8_t	[IN]	The length of the MAC in bytes
flags	uint32_t	[IN]	Bit0 – 0 encrypt / 1 decrypt

Returns:

If the decryption failed (MAC check failed), it returns an error code. Otherwise, it 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_AllocCtx

Prototype:

```
void* SHA1_AllocCtx (void)
```

Description:

Allocate memory for the SHA1 context.

Parameters:

None

Returns

Pointer to the context. NULL in case of error.

SHA1_FreeCtx

Prototype:

```
void SHA1_FreeCtx (void* pContext)
```

Description:

Free memory of the SHA1 context.

Parameters:

pContext void* [IN] pointer to the context

Return:

N/A

SHA1_Init ()

Prototype:

```
void SHA1_Init
(
void* context
);
```

Description: This function performs SHA1 initialization.

Parameters:

Name	Type	Direction	Description
context	void*	[IN]	Pointer to the SHA1 context

Returns:

None

SHA1_HashUpdate ()

Prototype:

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

Description: This function performs SHA1 algorithm.

Parameters:

Name	Type	Direction	Description
context	void*	[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
      (
      void* context,
     uint8_t* pOutput,
       );
```

Description: This function performs the final part of the SHA1 algorithm. The final hash is stored in the context structure.

Parameters:

Name	Type	Direction	Description
context	void*	[IN/OUT]	Pointer to the SHA1 context
pOutput	uint8_t*	[OUT]	Pointer to the output location

Returns:

None.

SHA1_Hash ()

Prototype:

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

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

Parameters:

Name	Type	Direction	Description
pData	uint8_t*	[IN]	Pointer to the plain text
numBytes	uint32_t	[IN]	Number of bytes
pOutput	uint8_t*	[OUT]	Pointer to the output location

Returns:

None.

SHA256_AllocCtx

Prototype:

```
void* SHA256_AllocCtx (void)
```

Description:

Allocate memory for the SHA256 context.

Parameters:

None

Return:

Pointer to the context. NULL in case of error.

SHA256_FreeCtx

Prototype:

```
void SHA256_FreeCtx (void* pContext)
```

Description:

Free memory of the SHA256 context.

Parameters:

pContext void* [IN] pointer to the context.

Return:

N/A

SHA256_Init ()

Prototype:

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

Description: This function performs SHA256 initialization.

Parameters:

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

Returns:

None

SHA256_HashUpdate ()

Prototype:

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

Description: This function performs SHA256 algorithm.

Parameters:

Name	Type	Direction	Description
Context	void*	[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
(
void* context,
uint8_t* pOutput
);
```

Description: This function performs the final part of the SHA256 algorithm. The final hash is stored in the context structure.

Parameters:

Name	Type	Direction	Description
Context	void*	[IN/OUT]	Pointer to the SHA256 context
pOutput	uint8_t*	[IN]	Pointer to the output location

Returns:

None

SHA256_Hash ()

Prototype:

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

Description: This function performs the entire SHA256 algorithm (initialize, update, finish) over the imput data. The final hash is stored in the context structure.

Parameters:

Name	Type	Direction	Description
pData	uint8_t*	[IN]	Pointer to the plain text
numBytes	uint32_t	[IN]	Number of bytes
pOutput	uint8_t*	[OUT]	Pointer to the output location

Returns:

None.

HMAC_SHA256_AllocCtx

Prototype:

```
void* HMAC_SHA256_AllocCtx (void)
```

Description:

Allocate memory for the HMAC context.

Parameters:

None

Return:

Pointer to the context. NULL in case of error.

HMAC_SHA256_FreeCtx

Prototype:

```
void HMAC_SHA256_FreeCtx (void* pContext)
```

Description:

Free memory of the HMAC context.

Parameters:

Context void* [IN] pointer to the context

Return:

N/A

HMAC_SHA256_Init ()

Prototype:

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

Description: This function performs HMAC initialization.

Parameters:

Name	Type	Direction	Description
ctx	void*	[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
(
void* ctx,
uint8_t* pData,
uint32_t numBytes
);
```

Description: This function performs HMAC algorithm based on SHA256.

Parameters:

Name	Type	Direction	Description
ctx	void*	[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
(
void * pContext,
uint8_t* pOutput
);
```

Description: This function performs the final part of the HMAC algorithm. The final hash is stored in the context structure.

Parameters:

Name	Type	Direction	Description
pContext	void * pContext	[IN/OUT]	Pointer to the HMAC context
pOutput	uint8_t*	[IN]	Pointer to the output location

Returns:

None

HMAC_SHA256 ()

Prototype:

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

Description: This function performs the entire HMAC algorithm (initialize, update, finish) over the input data. The final hash is stored in the context structure.

Parameters:

Name	Type	Direction	Description
pKey	uint8_t*	[IN]	Pointer to the HMAC key

Table continues on the next page...

Table continued from the previous page...

Name	Type	Direction	Description
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
pOutput	uint8_t*	[OUT]	Pointer to the output location

Returns:

None

3.13 Lists

3.13.1 Overview

This framework includes a general-purpose linked lists module. It implements common lists operations, such as

- 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

3.13.2 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.

3.13.3 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 an 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 at the start of the list.

Parameters:

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

Returns:

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

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; 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:

The 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.

3.13.4 Sample code

Linked list example

```
typedef struct userStruct_tag
{
    listElement_t anchor;
    uint32_t data;
}userStruct_t;
list_t list;
uint32_t userFunc(void)
{
    userStruct_t dataStruct, *dataStruct_ptr;
    ListInit(list, 0);
```

```
    dataStruct.data = 56;
    ListAddTail(list, (listElementHandle_t)dataStruct.anchor);
    dataStruct_ptr = (userStruct_t *)ListRemoveHead(list);
    return dataStruct_ptr->data;
}
```

3.14 Helper Functions Library

3.14.1 Overview

This framework provides a collection of features commonly used in embedded software centered on memory manipulation.

3.14.2 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; FALSE otherwise.

FLib_MemCmpToVal

Prototype:

```
bool_t FLib_MemCmpToVal (void* pAddr,
    void* val,
    uint32_t len
    );
```

Description: Compares a memory block to a byte value.

Parameters:

Name	Type	Direction	Description
pAddr	const void *	[IN]	Pointer to a memory block
Val	uint8_t	[IN]	Compare value
Len	uint32_t	[IN]	Number of bytes to copy

Returns:

If all the bytes from the memory block are equal to the compare value, the function returns TRUE; 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; 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.

3.15 Low-power Library

The low-power module (LPM) simplifies putting an NXP chip into the low-power or sleep modes to preserve battery life.

3.15.1 Overview

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 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. 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 configured at the application level. 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_Init()* function. The code for entering the low-power state must be placed in the Idle task or another low-priority application task that runs only after all the other (higher-priority) tasks finish all their work. 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.

Entering low-power state from a low-priority task

```
...
if( PWR_CheckIfDeviceCanGoToSleep() )
{
PWR_EnterLowPower();
```

```
}
...
```

3.15.2 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 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 does not save code space.

cPWR_DeepSleepMode

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

3.15.3 Low-power library API

PWR_Init

Prototype

```
void PWR_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().

3.15.4 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 a portion of the SRAM_U remains powered on. The 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: *cPWR_DeepSleepDurationMs* by default . This value can be changed at run-time.
    - Low-power timer (LPTMR) clock source: internal low-power oscillator (LPO).
    - Low-power timer (LPTMR) resolution : modified at run time to meet timeout value.
- Mode 3
  - MCU/Radio low-power modes:
    - MCU sleep mode is VLLS2 mode. Only a portion of the SRAM_U remains powered on. The 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: *cPWR_DeepSleepDurationMs* by default . This value can be changed at run-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) resolution : modified at run time to meet timeout value.
- Mode 4
  - MCU/Radio low-power modes:

    - MCU sleep mode is VLLS2 mode. Only a portion of the 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 :*cPWR_DeepSleepDurationMs* by default . This value can be changed at run-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 a portion of the 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 :*cPWR_DeepSleepDurationMs* by default . This value can be changed at run-time.
    - Low-power timer (LPTMR) clock source – internal low-power oscillator (LPO).
    - Low-power timer (LPTMR) resolution : modified at run time to meet timeout value.
- Mode 6
  - MCU/Radio low-power modes:
    - MCU sleep mode is VLLS2 mode. Only a portion of the 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: *cPWR_DeepSleepDurationMs* by default . This value can be changed at run-time.
    - Low-power timer (LPTMR) clock source: ERCLK32K (secondary external reference clock - 32.768 kHz crystal connected to RTC oscillator).
    - Low-power timer (LPTMR) resolution : modified at run time to meet timeout value.
- Mode 7
  - MCU/Radio low-power modes:
    - MCU sleep mode is VLLS2 mode. Only a portion of the 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 :*cPWR_DeepSleepDurationMs* by default . This value can be changed at run-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 a portion of the 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: *cPWR_DeepSleepDurationMs* by default . This value can be changed at run-time.
    - Low-power timer (LPTMR) clock source: internal low-power oscillator (LPO).

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 MCU is in WAIT mode. The clock on the radio is still running, but the transceiver is disabled.

cPWR_LPTMRClockSource (not applicable for QN908X MCU-based platforms)

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_DeepSleepDurationMs

The default deep sleep duration in milliseconds used by the LPM. For deep sleep modes 4, 7, 11, the value must be multiple of 1000 ms.

The duration of sleep can be changed at runtime using *PWR_SetDeepSleepTimeInMs()* or *PWR_SetDeepSleepTimeInSymbols()*.

cPWR_CallWakeupStackProcAfterDeepSleep (not applicable for QN908X MCU-based platforms)

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 (not applicable for QN908X MCU-based platforms)

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 (not applicable for QN908X MCU-based platforms)

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 (not applicable for QN908X MCU-based platforms)

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).

cPWR_CheckLowPowerTimers (not applicable for QN908X MCU-based platforms)

If enabled, the Low-Power module searches for the first Low-Power-Timer to expire and sets up the sleep duration as the minimum between the timer time-out value and the requested sleep duration.

3.15.5 Kinetis Wireless Dual Mode MKW4x Microcontrollers Low-power Library Overview

3.15.5.1 Overview

The dual mode Kinetis wireless microcontrollers, for example MKW41Z, 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, Generic FSK, or IEEE 802.15.4) and a DC-to-DC converter. These features of the SoC require a certain functionality of the low-power managing firmware modules.

3.15.5.2 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. Therefore, the system enters low-power on idle task, which runs only when there are no events for other tasks. The user must call the function PWR_EnterLowPower in the idle task as shown here:

```
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 stays awake until all software layers that called PWR_DisallowDeviceToSleep() call back PWR_AllowDeviceToSleep() and the idle task starts running. The MCU enters either a sleep or a deep sleep state depending on the type of the timers started. Low-power timers are the only timers that do not prevent the system from entering a deep sleep. If any other timers are started, the MCU will enter a sleep instead of a deep sleep state. Therefore, to enter deep sleep, stop all timers except the low-power timers. Be aware that functions, such as the LED_StartFlash, start timers which prevent the system from entering a deep sleep state.

3.15.5.3 Deep sleep modes

Seven low-power modes have been developed 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 the PWR_Configuration.h header file.

3.15.5.3.1 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 the on-board switches, by LPTMR timeout, or by BLE Link Layer wakeup interrupt (BLE_LL reference clock reaches wake up instance register)

using LLWU module. The LPTMR timer is used to measure the time that the MCU spends in deep sleep 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. The 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. The 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 is 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, the BLE Link Layer wakes up the entire SoC and the system resumes its activity. The two defines above can be found in the PWR_Configuration.h header file.

3.15.5.3.2 Deep sleep mode 2

This low-power mode was designed to be used when all stacks are idle.

In this mode, the MCU enters LLS2 and all enabled link layers remain idle. Depending on the value of the cPWR_RAM2_EnabledInLLS2 defined in the PWR_Configuration.h file, 16 kB of RAM (0x20000000- 0x20003fff if cPWR_RAM2_EnabledInLLS2=0) or 32 kB of RAM (0x20000000- 0x20007fff if cPWR_RAM2_EnabledInLLS2 = 1) are retained. It is application responsibility to allocate the variables that need to be retained over low-power in these areas. The SoC wakes up from this mode by the on-board switches, by DCDC power switch (when DCDC is in buck mode), or by LPTMR timeout using LLWU module. LPTMR timer is also used to measure the time MCU spends in deep sleep 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).

3.15.5.3.3 Deep sleep mode 3

This low-power mode was designed to be used when all stacks enabled for this platform are idle.

In this mode, the MCU enters LLS3 and all enabled link layers remain idle. All RAM is retained. The SoC wakes up from this mode by the on-board switches, by DCDC power switch (when DCDC is in buck mode), or by LPTMR timeout using LLWU module. The LPTMR timer is also used to measure the time that MCU spends in deep sleep 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).

3.15.5.3.4 Deep sleep mode 4

This is the lowest power mode. It was designed to be used when all stacks enabled for this platform are idle. In this mode, the MCU enters VLLS0/VLLS1 and all enabled link layers remain idle. The SoC wakes up from this mode by the on-board switches or by DCDC power switch (when DCDC is in buck mode) using LLWU module. No synchronization for low-power timers is made because 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, the MCU enters VLLS1 because 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 the POR circuit in VLLS0 making this deep sleep mode lowest power mode possible.

3.15.5.3.5 Deep sleep mode 5

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

In this mode, the MCU enters VLLS2 and all stacks enabled to run on this platform remain idle. A chip-dependent amount of upper RAM (4 kB for KW40X/16 kB or 32 kB for KW41Z) is retained. The SoC wakes up from this mode by on-board switches 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 the reset sequence.

For the KW35/KW36 family, depending on the value of the cPWR_RamRetentionInVLLS define, the device will preserve 16/32KB (VLLS2) or entire RAM (VLLS3). Also, there is an option to use a warm-boot sequence after the wake-up from VLLS2/3, to skip over the data initialization. Only the used peripherals that do not retain their state must be restored in this case. The timers are synchronized and all application data and state machines are preserved (see "Warm boot support" chapter).

3.15.5.3.6 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 maintains its state. Any module capable of producing an interrupt can wake up the MCU, such as on-board switches, DCDC power switch (when DCDC is in buck mode), LPTMR timeout, Radio Interrupt, UART, and so on. The LPTMR timer is also used to measure the time that the MCU spends in deep sleep to synchronize low-power timers at wakeup. The deep sleep timeout can be configured at compile time using cPWR_DeepSleepDurationMs define in the 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).

3.15.5.3.7 Deep sleep mode 7

This is a GenFSK specific low-power mode so it can be used only when cPWR_GENFSK_LL_Enable macro is set. The MCU enters LLS3 and GenFSK link layer must already be in DSM by the time this mode is entered. This can be achieved using PWR_GENFSK_EnterDSM(uint32_t dsmDuration) function. The GenFSK link layer enters DSM as soon as possible and sleeps for dsmDuration milliseconds. All RAM is retained. The SoC wakes up from this mode by the onboard switches, by LPTMR timeout, or by DSM exit wakeup interrupt (DSM timer finished counting dsmDuration) using LLWU module. If the MCU wakes up for other reason than DSM exit interrupt, the GenFSK link layer is also waken up as soon as possible. The LPTMR timer is used to measure the time that the MCU spends in deep sleep to synchronize low-power timers at wakeup. The LPTMR 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 LPTMR timeout in this mode is 65535000 ms (18.2h).

3.15.5.3.8 Deep sleep mode 8

This low-power mode is similar with the low-power mode 1. The main difference is that the MCU enters VLLS2 or VLLS3, depending on the amount of RAM preserved configured by the *cPWR_RamRetentionInVLLS* define.

Also, this low-power mode is intended to be used with a warm-boot sequence, to skip over the data initialization since the RAM is retained.

The SoC wakes up from this mode by the on-board switches, by LPTMR timeout, or by BLE Link Layer wakeup interrupt (BLE_LL reference clock reaches wake up instance register) using LLWU module. The LPTMR timer is used to measure the time that the MCU spends in deep sleep to synchronize low-power timers at wakeup .

Warm boot support

When the warm-boot is enabled, the application data, BLE stack data, Connectivity Framework is retained in RAM. Also, BLE LL and XCVR are in state retention in VLLS2/3, so no re-initialization is required. However, MCU general registers and peripherals

that are not in state retention need to be restored. To achieve this, the application must implement the following callbacks that the Low Power Manager module will call at low power entry and low power exit:

1. At VLLS entry to save application specific context.

   ```
   void PWR_RegisterVLLSEnterCallback(pfPWRCallBack_t vllsEnterCallback);
   ```

   In the application provided as example it saves:

   - VTOR
   - NVIC interrupt configuration
   - System clock gating control registers for enabled peripherals
   - System Tick

2. At warm boot to restore the application entry context and re-initialize modules.

   ```
   void PWR_RegisterWarmbootCallback(pfPWRCallBack_t warmBootCallback);
   ```

   It restores everything that was saved in the VLLS entry callback and re-initializes:

   - DCDC
   - Stack Timer
   - Switches

The Low Power Manager in the connectivity framework will handle the rest of the support:

- At VLLS2/3 entry in the LPM module, the CPU context is saved using setjmp.
- Startup sequence has been modified to skip over data initialization and execute a warm boot instead of normal boot at VLLS exit.
- Warm main executes the application callback on a dedicated warm boot stack.
- It then restores the CPU context with longjmp and execution is resumed.

The CPU start-up code was updated to check if this Reset is caused by a wake-up from VLLS2 or VLLS3 and if the warm-boot sequence is enabled (WARMBOOT_SEQUENCE = WARMBOOT_CONFIG stored), the program will skip over the RAM data initialization since it is preserved, and call warmmain().

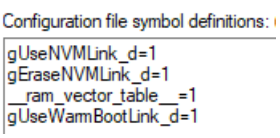
The warmmain() function will use a dedicated stack by default. The warm-boot stack should be in a RAM region which is not preserved in VLLS2 and has a default size of 1 KB.

The warm-boot config region must be located after the RAM vector table, at a location which is always preserved in VLLS2, and needs 8 bytes allocated.

When the warm-boot is enabled, the application stack is placed immediately after the program data, to be preserved in VLLS2.

By default, the Low Power Module will enable the warm-boot sequence with a dedicated stack (USE_WARMBOOT_SP=1) when low power modes 5 or 8 are enabled.

To enable the warm-boot config and stack for IAR IDE in linker file, the "gUseWarmBootLink_d=1" symbol must be added to the project linker configuration. The stack size value can be configured using the "__warmboot_stack_size__" linker symbol.



To enable warm-boot config and stack for MCUX, the WARMBOOT_CONFIG and WARMBOOT_STACK memory regions must be added in the memory config of the MCU settings.

Type	Name	Alias	Location	Size	Driver
Flash	PROGRAM_FLASH	Flash	0x0	0x7b800	FTFE_2K_PD.cfx
Flash	NVM_region	Flash2	0x7b800	0x4000	FTFE_2K_PD.cfx
Flash	FREESCALE_PROD_DATA	Flash3	0x7f800	0x800	FTFE_2K_PD.cfx
RAM	SRAM	RAM	0x200000c8	0xbf38	
RAM	RAM_VECTOR_TABLE	RAM2	0x20000000	0xc0	
RAM	WARMBOOT_CONFIG	RAM3	0x200000c0	0x8	
RAM	WARMBOOT_STACK	RAM4	0x1fffc000	0x400	

RTOS Support for Warm Boot

The warm-boot support is also available for FreeRTOS. In this case there is no need for a dedicated warm-boot stack (USE_WARMBOOT_SP=0), since the main stack (MSP) which is used by the interrupts is separated from the process stack (PSP) and can be used instead.

3.15.5.4 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_EnableDeepSleepMode_1      1
#define cPWR_EnableDeepSleepMode_2      1
#define cPWR_EnableDeepSleepMode_3      1
#define cPWR_EnableDeepSleepMode_4      1
#define cPWR_EnableDeepSleepMode_5      1
#define cPWR_EnableDeepSleepMode_6      1
#define cPWR_EnableDeepSleepMode_7      1
#define cPWR_EnableDeepSleepMode_8      1
```

Description

These defines were provided to save code size. If a particular deep sleep mode is not used, the corresponding define can be set to 0, causing it to be excluded from the build. To maintain compatibility with other low-power components, the PWR_EnterPowerOff(void) function was kept, but, in this implementation it puts the system in the lowest power mode possible, which is mode 4. Be aware that disabling mode 4 (cPWR_EnableDeepSleepMode_4 = 0), puts this function in an infinite loop.

Location

PWR_Configuration.h

Name

```
#define cPWR_DeepSleepDurationMs 30000
```

Description

Configures default deep sleep duration. It only has meaning for deep sleep modes 1, 2, 3, and 6.

Location

PWR_Configuration.h

Name

```
#define cPWR_BLE_LL_Enable FALSE
```

Description

Enables/disables BLE Link Layer Deep Sleep Mode functionalities.

Location

PWR_Configuration.h

Name

```
#define cPWR_GENFSK_LL_Enable FALSE
```

Description

Enables/disables GenFSK Link Layer Deep Sleep Mode functionalities.

Location

PWR_Configuration.h

Name

```
#define cPWR_Z_LL_Enable       FALSE
#define cPWR_ANT_LL_Enable     FALSE
```

Description

Enables/disables Z_LL, ANT_LL functionalities. None of these functionalities are currently implemented and these defines do not have any effect.

Location

PWR_Configuration.h

Name

```
#define cPWR_Zigbee_Enable FALSE
```

Description

This define enables/disables some legacy ZigBee stack-related checks before entering low-power. Do not enable it unless you have the legacy Zigbee stack running on this device.

Location

PWR_Configuration.h

Name

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

Description

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

Location

PWR_Configurations.h

Name

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

Description

The time period that the BLE Link Layer must wait after a system clock request to ensure the oscillator is running and is 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 the RF oscillator to stabilize the clock output after the oscillator is turned ON. In this period, the clock is assumed to be unstable and the controller does not turn on the clock to internal logic until this period is over. This means that the wake up from deep sleep mode must account for the delay before the wakeup instant.

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

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

(Warning: Maximum 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 configures deep sleep mode 4 to put the MCU in VLLS0 (when 1) or VLLS1 (when 0) because VLLS0 is only supported in bypass mode.

Location

PWR_Configurations.h

Name

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

Description

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

Location

PWR_Configurations.h

Name

```
#define gAllowDeviceToSleep_c 0
```

Description

Functions PWR_AllowDeviceToSleep/PWR_DisallowDeviceToSleep 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

Name

```
#define USE_WARMBOOT_SP 1
```

Description

Enable/Disable dedicated stack for the warm-boot sequence.

Location

PWR_Interface.h

Name

```
#define cPWR_RamRetentionInVLLS (2)
```

Description

Configure the amount of RAM to be retained in VLLS. Depending on this value, VLLS2 or VLLS3 deep sleep mode will be selected:

1. 16 K RAM retention (0x20000000- 0x20003fff): VLLS2 2
2. 32 K RAM retention (0x20000000- 0x20007fff): VLLS2 3
3. Full RAM retention: VLLS3

Location

PWR_Interface.h

Data type definitions

Name

```
typedef  union
{
    uint16_t AllBits;
    struct
    {
        uint8_t FromReset        :1;  /* Comming from Reset */
        uint8_t FromPSwitch      :1;  /* Wakeup by DCDC from Pswitch Pin */
        uint8_t FromKeyBoard     :1;  /* Wakeup by TSI/Push button interrupt */
        uint8_t FromLPTMR        :1;  /* Wakeup by LPTMR timer interrupt */
        uint8_t FromRadio        :1;  /* Wakeup by RTC timer interrupt */
        uint8_t FromBLE_LLTimer  :1;  /* Wakeup by BLE_LL Timer */
        uint8_t DeepSleepTimeout :1;  /* DeepSleep timer ran out */
        uint8_t SleepTimeout     :1;  /* Sleep timer ran out */
        uint8_t FromSerial       :1;  /* Wakeup by Serial RX */
    } Bits;
} PWRLib_WakeupReason_t;
```

Description

The 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. If the 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 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 in the primitives section.

Location

PWR_Interface.h

3.15.5.5 API Primitives

Prototype

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

Description

This function initializes all hardware modules involved in the 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. The valid values are 0, 1, 2, 3, 4, 5, and 6. If 0 is chosen, the system doesn't enter deep sleep.

Returns

The function returns FALSE if it receive as parameter a value above six; 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 the 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 the system awake, calls PWR_DisallowDeviceToSleep and the variable is incremented. As software layers/entities decide that they can enter deep sleep, they call PWR_AllowDeviceToSleep and the variable is decremented. Software layers/entities must not 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.

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

A call to this function must be placed in the idle task to put the system in low-power. First, this function checks any non low-power timers are started. If there are, the function tries to put the system to sleep. Otherwise, it tries to put the system to deep sleep. The 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. If all conditions are met, the system is 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 or five, 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 or 5 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 action. First, the function sets the wakeup instant received as a parameter in the 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 parameter represents the wakeup moment in regard to the 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 as 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 deepSleepTimeMs);
```

Description

The function sets the value of deep sleep duration. The function has meaning only for deep sleep mode 1, 2, 3, and 6.

Parameters

uint32_t deepSleepTimeMs: The new value of deep sleep duration. Upon entering deep sleep the value is truncated in regard to 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 is called just before entering low-power modes 1, 2, 3, 4, and 5. The callback must 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 called 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 is called just after exiting low-power modes 1, 2, 3, 4, and 5. For modes 4 and 5, it is called only if the system fails to enter low-power. Otherwise, the system exits from these modes through the 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, and so on) can be made here as well. DCDC_PrepareForContinuousMode should be the first function called from this callback.

Parameters

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

Returns

None

Name

PWR_GetTotalSleepDurationMS

Prototype

```
uint32_t PWR_GetTotalSleepDurationMS(void);
```

Description

Returns the duration of the deep sleep in milliseconds.

Name

PWR_ResetTotalSleepDuration

Prototype

```
void PWR_ResetTotalSleepDuration(void);
```

Description

Reset the deep sleep duration counter.

Name

PWR_RegisterVLLSEnterCallback

Prototype

```
void PWR_RegisterVLLSEnterCallback(pfPWRCallBack_t vllsEnterCallback);
```

Description

Register a function to be called just before entering VLLS mode.

Name

PWR_RegisterWarmbootCallback

Prototype

```
void PWR_RegisterWarmbootCallback(pfPWRCallBack_t warmBootCallback);
```

Description

Register a function to be called from warmmain() at warm-boot sequence, before restoring CPU context.

3.15.6 Low power library for QN908X MCU based platforms

QN908X MCU has a different low-power management unit than the Kinetis MCUs, so the low-power NXP Framework library had to be adapted to support it.

3.15.6.1 QN908x deep sleep modes

QN908x has only two deep sleep modes called power down 0 (PD0) and power down 1 (PD1). These are described in detail in the “UM11023 - QN908x user manual.pdf”. For both modes the needed RAM memory blocks are powered on (the user can configure which blocks stay powered on and which are powered down during deep sleep). CPU registers values can be retained during deep sleep (the user can also disable this feature).

- Power down 0

All peripheral clocks and all clock sources are switched off keeping only the 32 kHz clock running, RTC timer is still running. Arm CPU is in deep sleep mode, BLE controller is in deep sleep mode and it can wake-up the entire chip if BLE in case of BLE events.

Possible wake up sources: BLE Core, RTC, GPIO, CapSense, ACMP

Power down 1

All peripheral clocks and all clock sources are switched off, the 32 kHz clock is also switched off. No timer is powered on, board can wake up only by GPIO input (button press) or analog comparator external input.

Note that In this mode there is no deterministic way of waking the board up from the application layer. BLE core and RTC are powered down and cannot wake the chip up.

Possible wake up sources: GPIO, ACMP

3.15.6.2 Low-power library deep sleep modes mapping

For compatibility with all existing applications based on NXP Connectivity Framework, the QN9080x power down modes were mapped on the existing deep sleep modes 1 to 6 as follows.

Deep sleep mode 3 (PD0) is the default mode and it is initialized at compile time using the cPWR_DeepSleepMode macro.

- Deep sleep mode 1
  - Calling PWR_ChangeDeepSleepMode(1) will set the deep sleep mode to PD0. This mode can be used when BLE controller is active in any way (the controller will try to enter deep sleep during events).
- Deep sleep mode 2
  - The same as Deep sleep mode 1 – PD0.
- Deep sleep mode 3
  - The same as Deep sleep mode 1 – PD0.
- Deep sleep mode 4

Calling PWR_ChangeDeepSleepMode(4) will set the deep sleep mode to PD1. This mode should be used when BLE controller is inactive. All low power timers will be disabled/removed upon entering this mode.

- Please note that the chip will not enter PD1 if BLE controller is active in any way (scanning, advertising or connected) even if requested to do so. Instead the chip it will try to enter PD0 while BLE controller is active.
- Deep sleep mode 5 is undefined.
- Deep sleep mode 6 is undefined.

3.15.7 Implementing FreeRTOS Tickless low-power mode

To implement the FreeRTOS Tickless mode, the system must not use tasks that are always running. Tasks should wait for events or use the OS task delay API.

Also, the Connectivity Low-Power module must be enabled (define *cPWR_UsePowerDownMode* to 1) and the FreeRTOS *configUSE_TICKLESS_IDLE* option must be set to 2 (user-defined implementation).

This will cause the low-power handling from the app's idle task to run in a task critical section of FreeRTOS context.

Below is a user-defined implementation example of the *vPortSuppressTicksAndSleep()* function that is called by FreeRTOS. This implementation will replace the one from "fsl_tickless_systick.c", which is defined as weak.

```
void vPortSuppressTicksAndSleep ( TickType_t xExpectedIdleTime )
{
    PWRLib_WakeupReason_t wakeupReason;

    if ( PWR_CheckIfDeviceCanGoToSleep() )
    {
        PWR_SetDeepSleepTimeInMs(xExpectedIdleTime * portTICK_PERIOD_MS);
        PWR_ResetTotalSleepDuration();

        /* Enter Low Power */
        wakeupReason = PWR_EnterLowPower();

#if gKBD_KeysCount_c > 0
        /* Woke up on Keyboard Press */
        if(wakeupReason.Bits.FromKeyBoard)
        {
            KBD_SwitchPressedOnWakeUp();
        }
#endif
        /* Get actual deep sleep time, and converted to OS ticks */
        xExpectedIdleTime = PWR_GetTotalSleepDurationMS() / portTICK_PERIOD_MS;

        portENTER_CRITICAL();
        /* Update the OS time ticks. */
        vTaskStepTick ( xExpectedIdleTime );
```

```
        portEXIT_CRITICAL();
    }
    else
    {
        /* Enter MCU Sleep */
        PWR_EnterSleep();
    }
}
```

3.16 Mobile Wireless Systems Coexistence

3.16.1 Overview

The Mobile Wireless Systems (MWS) Coexistence module was modelled after the Bluetooth Core v4.1 and is used to allow the coexistence of multiple wireless protocols (such as Bluetooth and ZigBee stacks) on the same MCU and/or protocols on different MCUs (such as Bluetooth and Wi-Fi).

The MWS functionality is not used for the QN908X MCU-based systems. The QN9080 has a hardware implementation of the coexistence module. The only function used is MWS_CoexistanceEnable() and its only purpose is to enable the hardware module and route the proper input and output signals.

This API was intended for use by the link layers of the protocol stacks, but higher layers can also control the access to the resources.

For the on-chip coexistence on dual mode microcontrollers such as Kinetis KW40Z or KW41Z, the BLE Link Layer is implemented in silicon digital blocks, which gives it a priority over the IEEE 802.15.4 Link Layer (MAC), which is implemented in software. This means that the IEEE 802.15.4 can be active only in the inactive portions of the BLE. For this reason, the IEEE 802.15.4 PHY always checks the Bluetooth inactivity duration before starting a new sequence. If there is not enough time to complete the sequence, it notifies the upper layer that it could not access the channel.

For the inter-ICs (off-chip) coexistence, the module uses three signals to indicate RF activity and request access to the medium.

There are two supported protocols: a priority/status based protocol and a priority only based protocol.

Pin name	Direction	Description
RF_ACTIVE	Output	Signals when the transceiver becomes active. The signal is active high.
RF_STATUS_PRIO	Output	If a high-priority sequence is requested, the line will go high for 50us. If access is granted within 100 us, this line will consequently signal the RF activity type: goes low for RX and stays high for TX. If a low-priority sequence is requested, the line will only signal the RF activity type in a similar manner.
RF_DENY	Input	Signals if the access to the medium is granted or not. The signal logic is configurable by the software. Default is active high meaning that a low level means access granted.

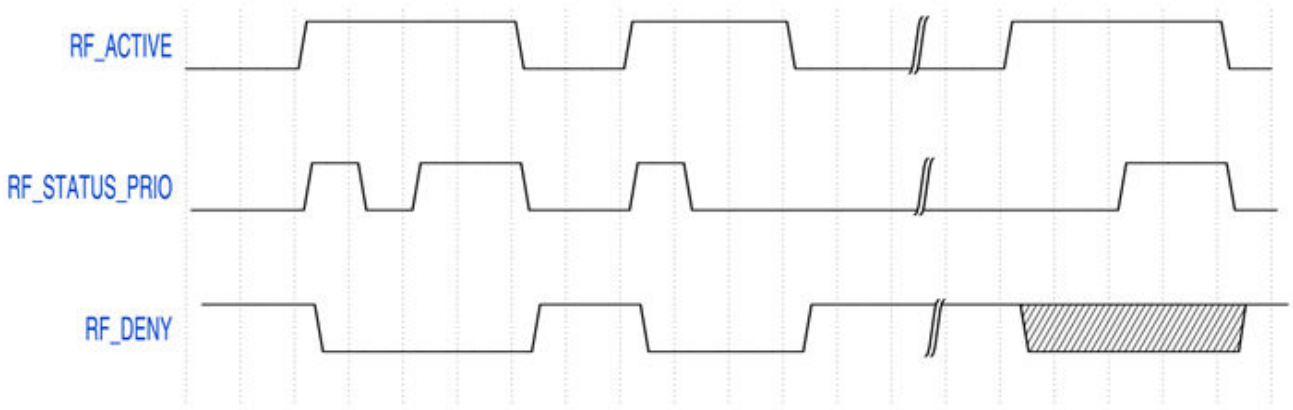


Figure 16. Inter ICs Coexistence signals for priority/status protocol

Pin name	Direction	Description
RF_ACTIVE	Output	Signals when the transceiver becomes active. The signal is active high.
RF__PRIO	Output	For a high-priority RF sequence, the line will go high. For a low-priority sequence, the line will go low.
RF_DENY	Input	Signals if the access to the medium is granted or not. The signal logic is configurable by the software. Default is active high meaning that a low level means access granted.

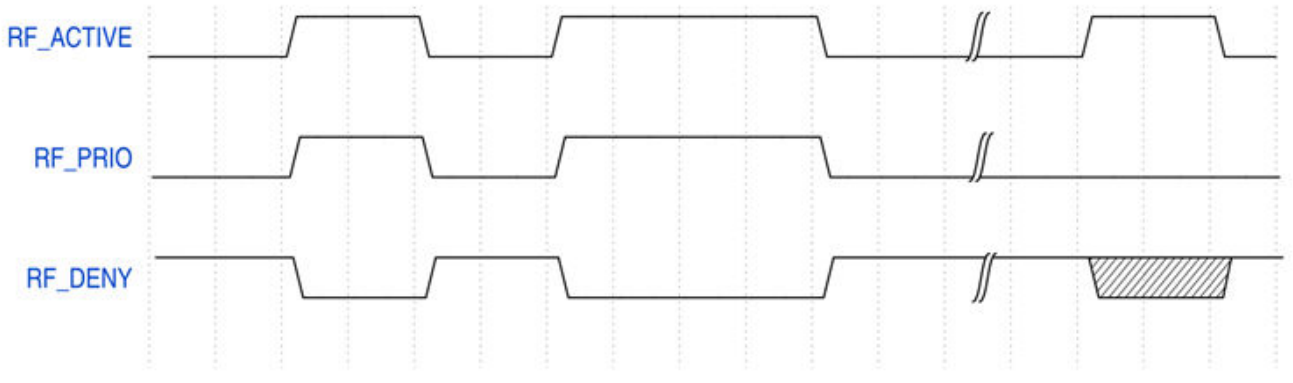


Figure 17. Inter ICs Coexistence signals for priority only protocol

On KW41Z, in the case of BLE coexistence, the RF_ACTIVE and RF_STATUS_PRIO pins are controlled by the hardware and cannot be changed. The RF_DENY pin is an interrupt-enabled GPIO and can be changed by the user.

On the same chip but for IEEE 802.15.4 coexistence, all the three pins can be configured by the user.

The most common coexistence cases are BLE/Wi-Fi and IEEE 802.15.4/Wi-Fi.

For KW2xD and MCR20A based solutions, all coexistence pins are microcontroller GPIOs. Any available GPIOs can be selected for the coexistence module.

Table 6. Coexistence pins assignment for priority/status based protocol

Platform	RF_ACTIVE	RF_STATUS	RF_DENY
MKW41Z	ANT_B / ANT_A	RX_SWITCH	any GPIO
MKW2xD & MCR20	any GPIO	any GPIO	any GPIO

Table 7. Coexistence pins assignment for priority-only based protocol

Platform	RF_ACTIVE	RF_STATUS	RF_DENY
MKW41Z	ANT_B	ANT_A	any GPIO
MKW2xD & MCR20	any GPIO	any GPIO	any GPIO

3.16.2 Constant Macro Definitions

gMWS_Enabled_d

```
#ifndef gMWS_Enabled_d
#define gMWS_Enabled_d 0
#endif
```

Description:

Enables/disables the coexistence of on-chip wireless protocols.

gMWS_UsePrioPreemption_d

```
#ifndef gMWS_UsePrioPreemption_d
#define gMWS_UsePrioPreemption_d 1
#endif
```

Description:

Enable the preemption of another protocol of a lower priority.

gMWS_UseCoexistence_d

```
#ifndef gMWS_UseCoexistence_d
#define gMWS_UseCoexistence_d 0
#endif
```

Description:

Enable/disables the coexistence between wireless protocols from different ICs.

gMWS_BLEPriority_d

```
#ifndef gMWS_BLEPriority_d
#define gMWS_BLEPriority_d 0
#endif
```

Description:

The priority of the BLE protocol. A lower number represents higher priority.

gMWS_802_15_4Priority_d

```
#ifndef gMWS_802_15_4Priority_d
#define gMWS_802_15_4Priority_d 1
#endif
```

Description:

The priority of the IEEE 802.15.4 protocol. A lower number represents higher priority.

gMWS_GFSKPriority_d

```
#ifndef gMWS_GFSKPriority_d
#define gMWS_GFSKPriority_d 2
#endif
```

Description:

The priority of the Generic FSK protocol. A lower number represents higher priority.

gMWS_ANTPriority_d

```
#ifndef gMWS_ANTPriority_d
#define gMWS_ANTPriority_d 3
#endif
```

Description:

The priority of the ANT+ protocol. A lower number represents higher priority.

gMWS_CoexPrioSignalTime_d

```
#ifndef gMWS_CoexPrioSignalTime_d
#define gMWS_CoexPrioSignalTime_d   (40) /* us */
#endif
```

Description:

The duration of the priority signal on the RF_STATUS_PRIO pin, in microseconds. The signal asserts at the same time as RF_ACTIVE.

gMWS_CoexRfActiveAssertTime_d

```
#ifndef gMWS_CoexRfActiveAssertTime_d
        #define gMWS_CoexRfActiveAssertTime_d (100) /* us */
        #endif
```

Description:

The RF_ACTIVE pin is asserted gMWS_CoexRfActiveAssertTime_d microseconds before the Tx/Rx begins.

gMWS_CoexConfirmWaitTime_d

```
#ifndef gMWS_CoexConfirmWaitTime_d
#define gMWS_CoexConfirmWaitTime_d 90
#endif
```

Description:

The time to wait for the RF_DENY pin to be deasserted, in microseconds. If RF_DENY is deasserted, access to the medium is granted. Else the access is not granted.

gMWS_CoexRfDenyActiveState_d

```
#ifndef gMWS_CoexRfDenyActiveState_d
#define gMWS_CoexRfDenyActiveState_d 1
#endif
```

Description:

The polarity of the RF_DENY pin:

1 – Access denied when logic one;

0 – Access denied when logic zero.

gMWS_CoexRxPrio_d

```
#ifndef gMWS_CoexRxPrio_d
#define gMWS_CoexRxPrio_d (gMWS_HighPriority)
#endif
```

Description:

The default priority of RX sequences. See mwsRfSeqPriority_t type definition for allowed values.

gMWS_CoexTxPrio_d

```
#ifndef gMWS_CoexTxPrio_d
#define gMWS_CoexTxPrio_d (gMWS_HighPriority)
#endif
```

Description:

The default priority of TX sequences. See mwsRfSeqPriority_ttype definition for allowed values.

gMWS_Coex_Model_d

```
#ifndef gMWS_Coex_Model_d
#define gMWS_Coex_Model_d gMWS_Coex_Status_Prio_d
#endif
```

Description:

Used to define the coexistence model to be supported. It can be one of the following:

```
gMWS_Coex_Status_Prio_d or gMWS_Coex_Prio_Only_d
```

gMWS_Coex_AccessGrantedByReq_d

```
#ifndef gMWS_Coex_AccessGrantedByReq_d
#define gMWS_Coex_AccessGrantedByReq_d    1
#endif
```

Description:

This define specifies if it is necessary to assert the RF_ACTIVE line first to get the access grant. Most of the masters keep the RF_DENY line asserted by default, and de-assert it only in consequence of RF_ACTIVE line being asserted.

gMWS_Coex_UseRFNotAllowed_

```
#ifndef gMWS_Coex_UseRFNotAllowed_d
#define gMWS_Coex_UseRFNotAllowed_d    0
#endif
```

Description

Some MCUs have hardware capabilities to manage the RF_DENY line. This define specifies whether the RF_DENY line is managed by hardware (1) or by software using an interrupt enabled GPIO (0).

gMWS_FsciEnabled_d

```
#ifndef gMWS_FsciEnabled_d
#define gMWS_FsciEnabled_d (0)
#endif
```

Description:

Enables / disables MWS FSCI commands

1 – MWS FSCI commands are enabled

0 – MWS FSCI commands are disabled

3.16.3 User defined data type definitions

mwsStatus_t

```
typedef enum
{
    gMWS_Success_c = 0,
    gMWS_Denied_c,
    gMWS_InvalidParameter_c,
    gMWS_Error_c
}mwsStatus_t;
```

Description:

MWS error codes.

mwsProtocols_t

```
typedef enum
{
    gMWS_BLE_c,
    gMWS_802_15_4_c,
    gMWS_GFSK_c,
    gMWS_ANT_c,
    gMWS_None_c /*! must be the last item */
}mwsProtocols_t;
```

Description:

Protocols supported by MWS.

mwsEvents_t

```
typedef enum
{
    gMWS_Init_c,
    gMWS_Idle_c,
```

```
    gMWS_Active_c,
    gMWS_Release_c,
    gMWS_Abort_c,
    gMWS_GetInactivityDuration_c,
}mwsEvents_t;
```

Description:

Events used by MWS to signal the protocol stacks.

pfMwsCallBack_t

```
typedef uint32_t ( *pfMwsCallback ) ( mwsEvents_t event );
```

Description:

Callback used by MWS to notify a protocol stack when other protocol stack change its state.

mwsRfState_t

```
typedef enum
{
    gMWS_IdleState_c,
    gMWS_RxState_c,
    gMWS_TxState_c
}mwsRfState_t;
```

Description:

The state of the transceiver.

mwsRfSeqPriority_t

```
typedef enum
{
    gMWS_LowPriority_c,
    gMWS_HighPriority_c
}mwsRfSeqPriority_t;
```

Description:

The RF sequence priority.

mwsPinInterruptMode_t

```
typedef enum
{
    gMWS_PinInterruptFallingEdge_c,
    gMWS_PinInterruptRisingEdge_c,
    gMWS_PinInterruptEitherEdge_c
} mwsPinInterruptMode_t;
```

Description:

Pin interrupt modes used by RF_DENY pin (applicable only for “priority-only” coexistence model).

3.16.4 API Primitives

MWS_Register

Prototype:

```
mwsStatus_t MWS_Register (mwsProtocols_t protocol, pfMwsCallback cb);
```

Description: Register a protocol stack to MWS.

Parameters:

Name	Type	Direction	Description
protocol	mwsProtocol_t	[IN]	The protocol to be registered.
cb	pfMwsCalback_t	[IN]	The callback function called by MWS to signal events to the protocol stacks

Returns:

Status code.

MWS_Acquire

Prototype:

```
mwsStatus_t MWS_Acquire (mwsProtocols_t protocol, uint8_t force);
```

Description: Try to acquire the resource (XCVR).

Parameters:

Name	Type	Direction	Description
protocol	mwsProtocol_t	[IN]	The protocol trying to acquire the resources.
force	uint8_t	[IN]	Set to TRUE if the current protocol should be preempted

Returns:

Status code.

MWS_Release

Prototype:

```
mwsStatus_t MWS_Release (mwsProtocols_t protocol);
```

Description: Release the access to the resource (XCVR).

Parameters:

Name	Type	Direction	Description
protocol	mwsProtocol_t	[IN]	The protocol releasing the resources.

Returns:

Status code.

MWS_SignalIdle

Prototype:

```
mwsStatus_t MWS_SignalIdle (mwsProtocols_t protocol);
```

Description: Signals other protocols that the XCVR is now idle.

Parameters:

Name	Type	Direction	Description
protocol	mwsProtocol_t	[IN]	The protocol signaling the idle state.

Returns:

Status code.

MWS_Abort

Prototype:

```
mwsStatus_t MWS_Abort (void);
```

Description: Force the XCVR to idle. The active protocol aborts any ongoing sequence.

Parameters:

None

Returns:

Status code.

MWS_GetInactivityDuration

Prototype:

```
uint32_t MWS_GetInactivityDuration (mwsProtocols_t currentProtocol);
```

Description: Returns the minimum idle time of the registered protocol stacks in microseconds.

Because part of the BLE functionality is implemented in hardware, other protocols can use this API to perform Over-the-Air activity during the BLE idle period.

Parameters:

Name	Type	Direction	Description
currentProtocol	mwsProtocol_t	[IN]	The protocol asking for the inactivity duration.

Returns:

0xFFFFFFFF if the XCVR is idle. All protocol stacks are idle.

0x00 if the XCVR is used by another protocol.

Else, the number of microseconds until another protocol needs access to the XCVR.

MWS_GetActiveProtocol

Prototype:

```
mwsProtocols_t MWS_GetActiveProtocol (void);
```

Description: Returns the protocol that is currently using the XCVR.

Parameters:

None

Returns:

The active protocol.

MWS_CoexistenceInit

Prototype:

```
mwsStatus_t MWS_CoexistenceInit(void *rfDenyPin, void *rfActivePin, void *rfStatusPin);
```

Description: Initialize the wireless coexistence module for protocols on different ICs.

If one of the pins is controlled by hardware, the pointer to the configuration structure should be NULL.

Parameters:

Name	Type	Direction	Description
rfDenyPin	void *	[IN]	Pointer to a GPIO input pin config. Set to NULL if pin is controlled by Hardware.
rfActivePin	void *	[IN]	Pointer to a GPIO output pin config. Set to NULL if pin is controlled by Hardware.
rfStatusPin	void *	[IN]	Pointer to a GPIO output pin config. Set to NULL if pin is controlled by Hardware.

Returns:

Status code.

MWS_CoexistenceRegister

Prototype:

```
mwsStatus_t MWS_CoexistenceRegister (mwsProtocols_t protocol, pfMwsCallback cb);
```

Description: Register a protocol stack to the inter ICs coexistence module.

Parameters:

Name	Type	Direction	Description
protocol	mwsProtocol_t	[IN]	The protocol to be registered
cb	pfMwsCalback_t	[IN]	The callback function called by MWS to signal events to the protocol stack.

Returns:

Status code.

MWS_CoexistenceDenyState

Prototype

```
uint8_t MWS_CoexistenceDenyState(void)
```

Description

This function can be used to get the current status of the RF_DENY line.

Parameters:

None

Returns:

The status of the RF_DENY line: gMWS_CoexRfDenyActiveState_d or !gMWS_CoexRfDenyActiveState_d.

MWS_CoexistenceSetPriority

Prototype:

```
void MWS_CoexistenceSetPriority(mwsRfSeqPriority_t rxPrio, mwsRfSeqPriority_t txPrio);
```

Description: Set the priority of the RF sequences: TX, RX

Parameters:

Name	Type	Direction	Description
rxPrio	mwsRfSeqPriority_t	[IN]	The priority of the RX sequence
txPrio	mwsRfSeqPriority_t	[IN]	The priority of the TX sequence

Returns:

None.

MWS_CoexistenceRequestAccess

Prototype:

```
mwsStatus_t MWS_CoexistenceRequestAccess(mwsRfState_t newState);
```

Description: Request permission to access the medium for a specific RF sequence. The RF_ACTIVE and RF_STATUS pins will be set accordingly.

Parameters:

Name	Type	Direction	Description
newState	mwsRfState_t	[IN]	The RF sequence type

Returns:

Status code.

MWS_CoexistenceChangeAccess

Prototype:

```
mwsStatus_t MWS_CoexistenceChangeAccess (mwsRfState_t newState);
```

Description: Change the RF sequence type (RX->TX or TX->RX). The RF_STATUS pin state will be updated.

Parameters:

Name	Type	Direction	Description
newState	mwsRfState_t	[IN]	The new RF sequence type

Returns:

Status code.

MWS_CoexistenceReleaseAccess

Prototype:

```
void MWS_CoexistenceReleaseAccess(void);
```

Description: Release access to the medium. The RF_ACTIVE and RF_STATUS pins will transition to idle state.

Parameters:

None

Returns:

None

MWS_CoexistenceEnable

Prototype:

```
void MWS_CoexistenceEnable(void);
```

Description: Enables the coexistence signals.

Parameters:

None

Returns:

None

MWS_CoexistenceDisable

Prototype:

```
void MWS_CoexistenceDisable(void);
```

Description: Disables the coexistence signals.

Parameters:

None

Returns:

None

MWS_FsciInit

Prototype:

```
void MWS_FsciInit(void);
```

Description: Initialize the MWS FSCI functionality.

Parameters:

None

Returns:

None

MWS_FsciMsgHandler

Prototype:

```
void MWS_FsciMsgHandler(void* pData, void* param, uint32_t fsciInterface);
```

Description: Initialization function that registers MWS opcode group and the corresponding message handler.

Parameters:

Name	Type	Direction	Description
pData	void*	[IN]	Pointer to message data
param	void*	[IN]	Pointer to additional parameters (if any)
fsciInterface	uint32_t	[IN]	FSCI interface used

Returns:

None

3.17 LED

3.17.1 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
- RGB dimming, if an RGB LED is available and connected to PWM channels.

The flash and blip features use a timer from the Timers' Manager module. If the RGB dimming is enabled, another separate timer is allocated from Timer Manager module.

3.17.2 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 gLED_InvertedMode_d             FALSE
```

Description:

Specifies if LEDs are operated in inverted mode. When the GPIO output is high, the LED is on. This is defined as TRUE for the QN908x MCU-based platforms.

Name:

```
#define LED1                            0x01
#define LED2                            0x02
#define LED3                            0x04
#define LED4                            0x08
#define LED_ALL                         0x0F
#define LED1_FLASH                      0x10
#define LED2_FLASH                      0x20
#define LED3_FLASH                      0x40
#define LED4_FLASH                      0x80
#define LED_FLASH_ALL                   0xF0
```

Description:

LEDs mapping.

Name:

```
#define gLEDsOnTargetBoardDefault_c     4
```

Description:

Specifies the default number of LEDs available on the board. The LED module supports up to 4 LEDs.

Name:

```
#define gLedRgbEnabled_d               FALSE
```

Description:

Specifies if RGB led is supported by the LED module.

Name:

```
#define gLedColorWheelEnabled_d    TRUE
```

Description:

Specifies if RGB led supports color wheel.

Name:

```
#define gRgbLedDimmingEnabled_d    FALSE
```

Description:

Enables/disables the RGB led dimming.

Name:

```
#define gRgbLedDimDefaultInterval_c    3
```

Description:

Specifies the default dimming interval expressed in seconds.

Name:

```
#define gRgbLedDimMaxInterval_c       10
```

Description:

Specifies the maximum supported dimming interval expressed in seconds.

Name:

```
#define gRgbLedDimTimerTimeout_c      50
```

Description:

Specifies the dimming resolution expressed in milliseconds.

3.17.3 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().

3.17.4 API primitives

TurnOnLeds ()

Prototype:

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

Description: Turns on all LEDs.

Parameters:

None.

Returns:

None.

SerialFlashing ()

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 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 the 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 one or more LEDs flashing.

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 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 hexadecimal 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 hexadecimal 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.

LED_SetRgbLed

Prototype:

```
extern void LED_SetRgbLed(LED_t LEDNr,uint16_t redValue,uint16_t greenValue, uint16_t blueValue);
```

Description:

Sets a specified value on the RGB LED. The value on each LED has a valid range from 0 to PWM_MODULE_MAX_DUTY_CYCLE_c.

Parameters:

Name	Type	Direction	Description
LEDNr	LED_t	[IN]	LED(s) to operate
redValue	uint16_t	[IN]	Value for RED color
greenValue	uint16_t	[IN]	Value for GREEN color
blueValue	uint16_t	[IN]	Value for BLUE color

Returns:

None.

LED_StartColorWheel

Prototype:

```
void LED_StartColorWheel(LED_t LEDNr, uint16_t periodMs);
```

Description:

Starts RGB LED color wheel.

Parameters:

Name	Type	Direction	Description
LEDNr	LED_t	[IN]	LED(s) to operate
periodMs	uint16_t	[IN]	Period to the next color in milliseconds

Returns:

None.

LED_RgbDimOut

Prototype:

```
bool_t LED_RgbDimOut(void);
```

Description:

Dim (fade) out the RGB LED.

Parameters:

None

Returns:

TRUE if no dimining is ongoing and the command has been accepted, FALSE otherwise.

LED_RgbDimIn

Prototype:

```
bool_t LED_RgbDimIn(void);
```

Description:

Dim (fade) in the RGB LED.

Parameters:

None

Returns:

TRUE if no dimining is ongoing and the command has been accepted, FALSE otherwise.

LED_RgbSetDimInterval

Prototype:

```
void LED_RgbSetDimInterval(uint8_t dim_interval);
```

Description:

Sets the RGB LED dimming interval, in seconds.

Parameters:

Name	Type	Direction	Description
dim_interval	uint8_t	[IN]	Dimming interval

Returns:

None

LED_ExitLowPower

Prototype:

```
void LED_ExitLowPower(void);
```

Description: Led exit low-power mode.

Parameters:

None.

Returns:

None.

3.18 Keyboard

3.18.1 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 an interrupt context. The keyboard uses a timer from the Timers' Manager module for debouncing, short/long press, and hold detection.

3.18.2 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 is returned after this number of milliseconds if pressed. Ensure this constant is long enough for debounce time.

3.18.3 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.

3.18.4 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; FALSE otherwise.

3.19 GPIO Adapter

3.19.1 Overview

The module covers GPIO initialization and interrupt management. One interrupt vector is available for the entire GPIO port. This module allows the installation of an ISR callback 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.

3.19.2 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 when registering ISR callbacks.

3.19.3 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;
```

Description:

Defines the error codes returned by the API.

Name:

```
typedef enum gpioPort_tag
{
    gpioPort_A_c = 0,
    gpioPort_B_c = 1,
    gpioPort_C_c = 2,
    gpioPort_D_c = 3,
    gpioPort_E_c = 4,
    gpioPort_Invalid_c = 5
}gpioPort_t;
```

Description:

Defines MCU ports identification.

Name:

```
typedef enum portPull_tag
{
#if (FSL_FEATURE_SOC_INTMUX_COUNT <= 0) && (FSL_FEATURE_SOC_SYSCON_COUNT > 0)
    pinPull_Disabled_c  = 0U,
    pinPull_Down_c      = 1U,
    pinPull_Up_c        = 2U,
#else
    pinPull_Down_c      = 0U,
    pinPull_Up_c        = 1U,
    pinPull_Disabled_c  = 2U,
#endif
    pinPull_Invalid_c   = 3U,
} pinPullSelect_t;
```

Description:

Defines input pins pull configuration.

Name:

```
typedef enum portInterrupt_tag {
    pinInt_Disabled_c = 0x0U,
    pinDma_RisingEdge_c = 0x1U,
    pinDma_FallingEdge_c = 0x2U,
    pinDma_EitherEdge_c = 0x3U,
    pinInt_LogicZero_c = 0x8U,
    pinInt_RisingEdge_c = 0x9U,
    pinInt_FallingEdge_c =0xAU,
    pinInt_EitherEdge_c = 0xBU,
    pinInt_LogicOne_c = 0xCU,
    pinInt_Invalid_c = 0xDU
} pinInterrupt_t;
```

Description:

Defines input pins interrupt event configuration.

Name:

```
typedef enum
{
#if (FSL_FEATURE_SOC_INTMUX_COUNT <= 0) && (FSL_FEATURE_SOC_SYSCON_COUNT > 0)
    pinMux_Gpio_c = 0U,
    pinMux_Alt1_c = 1U,
#else
    pinMux_PinDisabledOrAnalog_c = 0U,
    pinMux_Gpio_c = 1U,
    pinMux_Alt7_c = 7U,
#endif
    pinMux_Alt2_c = 2U,
    pinMux_Alt3_c = 3U,
    pinMux_Alt4_c = 4U,
    pinMux_Alt5_c = 5U,
    pinMux_Alt6_c = 6U,
} pinMux_t;
```

Description:

Defines pins alternate functions.

Name:

```
typedef enum portSlewRate_tag {
    pinSlewRate_Fast_c = 0U,
    pinSlewRate_Slow_c = 1U,
    pinSlewRate_Invalid_c = 2U
} pinSlewRate_t;
```

Description:

Defines pins slew rate configuration.

Name:

```
typedef enum portDriveStrength_tag {
    pinDriveStrength_Low_c  = 0U,
    pinDriveStrength_High_c = 1U,
#if (FSL_FEATURE_SOC_INTMUX_COUNT <= 0) && (FSL_FEATURE_SOC_SYSCON_COUNT > 0)
```

```
  /* Extra and Full are only valid for QN9080 Port A, pins 6,11,19,26 and 27 */
    pinDriveStrength_Extra_c= 2U,
    pinDriveStrength_Full_c = 3U, /*!< High + Extra strength is configured. */
#endif
    pinDriveStrength_Invalid_c,
} pinDriveStrength_t;
```

Description:

Defines pins drive strength configuration.

Name:

```
typedef struct gpioInputPinConfig_tag
{
    gpioPort_t gpioPort;
    uint8_t gpioPin;
    pinPullSelect_t pullSelect;
    pinInterrupt_t interruptSelect;
}gpioInputPinConfig_t;
```

Description:

Defines input pins configuration.

Name:

```
typedef struct gpioOutputPinConfig_tag
{
    gpioPort_t gpioPort;
    uint8_t gpioPin;
    bool_t outputLogic;
    pinSlewRate_t slewRate;
    pinDriveStrength_t driveStrength;
}gpioOutputPinConfig_t;
```

Description:

Defines output pins configuration.

3.19.4 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 specified GPIO.

Parameters:

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

Returns:

The error code.

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.

GpioInputPinInit

Prototype:

```
bool_t GpioInputPinInit
(
const gpioInputPinConfig_t* pInputConfig,
uint32_t noOfElements
);
```

Description: Initializes input pins according to values in the input pins configuration structure array.

Parameters:

Name	Type	Direction	Description
pInputConfig	const gpioInputPinConfig_t*	[IN]	The address of the configuration structure array.
noOfElements	uint32_t	[IN]	Number of elements of the configuration structure array.

Returns:

TRUE for succesful initialization.

FALSE for incorrect parameters.

GpioOutputPinInit

Prototype:

```
bool_t GpioOutputPinInit
(
const gpioOutputPinConfig_t* pOutputConfig,
uint32_t noOfElements
);
```

Description: Initializes output pins according to values in the output pins configuration structure array.

Parameters:

Name	Type	Direction	Description
pOutputConfig	const gpioOutputPinConfig_t*	[IN]	The address of the configuration structure array.
noOfElements	uint32_t	[IN]	Number of elements of the configuration structure array.

Returns:

TRUE for successful initialization.

FALSE for incorrect parameters.

GpioReadPinInput

Prototype:

```
uint32_t GpioReadPinInput
(
const gpioInputPinConfig_t* pInputConfig
);
```

Description: Reads the logical level of the input pin described by the input pin configuration structure.

Parameters:

Name	Type	Direction	Description
pInputConfig	const gpioInputPinConfig_t*	[IN]	The address of the configuration structure array.

Returns:

Logical level of the input pin.

GpioIsPinIntPending

Prototype:

```
uint32_t GpioIsPinIntPending
(
const gpioInputPinConfig_t* pInputConfig
);
```

Description: Checks if the interrupt flag of the input pin described by the input pin configuration structure is set.

Parameters:

Name	Type	Direction	Description
pInputConfig	const gpioInputPinConfig_t*	[IN]	The address of the configuration structure array.

Returns:

The value of the interrupt flag of the input pin.

GpioClearPinIntFlag

Prototype:

```
void GpioClearPinIntFlag
(
const gpioInputPinConfig_t* pInputConfig
);
```

Description: Clears the interrupt flag of the input pin described by the input pin configuration structure.

Parameters:

Name	Type	Direction	Description
pInputConfig	const gpioInputPinConfig_t*	[IN]	The address of the configuration structure array.

Returns:

None.

GpioSetPinOutput

Prototype:

```
void GpioSetPinOutput
(
const gpioOutputPinConfig_t* pOutputConfig
);
```

Description: Set the logical level of the output pin described by the output pin configuration structure to 1.

Parameters:

Name	Type	Direction	Description
pOutputConfig	const gpioOutputPinConfig_t*	[IN]	The address of the configuration structure array.

Returns:

None.

GpioClearPinOutput

Prototype:

```
void GpioClearPinOutput
(
const gpioOutputPinConfig_t* pOutputConfig
);
```

Description: Set the logical level of the output pin described by the output pin configuration structure to 0.

Parameters:

Name	Type	Direction	Description
pOutputConfig	const gpioOutputPinConfig_t*	[IN]	The address of the configuration structure array.

Returns:

None.

GpioTogglePinOutput

Prototype:

```
void GpioTogglePinOutput
(
const gpioOutputPinConfig_t* pOutputConfig
);
```

Description: Toggles the logical level of the output pin described by the output pin configuration structure.

Parameters:

Name	Type	Direction	Description
pOutputConfig	const gpioOutputPinConfig_t*	[IN]	The address of the configuration structure array.

Returns:

None.

GpioSetPinMux

Prototype:

```
void GpioSetPinMux
(
gpioPort_t gpioPort,
uint8_t gpioPin,
pinMux_t pinMux
);
```

Description: Sets the alternate function of the pin.

Parameters:

Name	Type	Direction	Description
gpioPort	gpioPort_t	[IN]	GPIO port identifier
gpioPin	uint8_t	[IN]	Pin number(0 to 31)
pinMux	pinMux_t	[IN]	Alternate function to be set

Returns:

None.

GpioSetPinPullMode

Prototype:

```
void GpioSetPinPullMode
(
gpioPort_t gpioPort,
uint8_t gpioPin,
pinPullSelect_t pullSelect);
```

Description: Sets the pull enable and pull select attributes of the pin.

Parameters:

Name	Type	Direction	Description
gpioPort	gpioPort_t	[IN]	GPIO port identifier
gpioPin	uint8_t	[IN]	Pin number(0 to 31)
pullSelect	pinPullSelect_t	[IN]	Selects pull mode

Returns:

None.

GpioSetPassiveFilter

Prototype:

```
void GpioSetPassiveFilter
(
gpioPort_t gpioPort,
uint8_t gpioPin,
bool_t enable
);
```

Description: Enables/disables pin passive filter.

Parameters:

Name	Type	Direction	Description
gpioPort	gpioPort_t	[IN]	GPIO port identifier
gpioPin	uint8_t	[IN]	Pin number(0 to 31)
enable	bool_t	[IN]	Passive filter enable value to be set.

Returns:

None.

3.20 Kinetis DCDC Driver Reference

3.20.1 Overview

The DCDC converter module is a switching mode DC-DC converter supporting Buck, Boost, and Bypass modes. The DCDC converter produces two switching outputs in both the Buck and Boost modes. They are VDD1p45 (KW40Z)/VDD1p5 (KW41Z) (referred to as VDDMCU) and VDD1p8, which can produce up to 25 mA and 42.5 mA continuous output current respectively.

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

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

In Bypass mode, the DCDC converter is disabled. Individual supply signals of KW4xZ need to be supplied with regulated supply accordingly.

The DCDC operating mode is hardware selected through configuration jumpers.

The DCDC driver purpose is to ensure that the DCDC operates properly.

At DCDC driver initialization (DCDC_Init), the user passes DCDC operating parameters to the driver. The operating parameters are 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 chosen DCDC operating mode is Bypass mode, the driver takes no further action. Otherwise, the driver starts a low-power interval timer (the timer interval is the one received in the configuration structure at DCDC_Init) 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 depending on the battery voltage and user desired values, sets these values in the appropriate registers and then calls the application callback to notify the user about the DCDC status.

For example, let’s suppose 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 VDD1p8 power line and 1.45 V for the VDDMCU 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 VDDMCU power line since in Boost mode the output voltages must be at least 50 mV above the battery voltage.

3.20.2 Application Programming Interface

The 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 initialized (DCDC_init function starts a low-power interval timer. The TMR_Init() function only runs the first time it is called.

The DCDC driver initializes and uses the ADC driver, which can cause a conflict if other parts of the program also use the ADC driver or the ADC module.

3.20.3 Configuration Macros

Name:

```
#define gDCDC_Enabled_d 0
```

Description:

Enables/disables DCDC-related code and variables.

Location:

DCDC.h

3.20.4 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 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 and the driver must be notified about the selected configuration.

Location:

DCDC.h

Name:

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

Description:

Defines the events that the DCDC driver notifies the application about:

- gDCDC_Event_VBatOutOfRange_c: VDCDC_IN (battery voltage) is no longer in range defined by the user in the DCDC configuration structure passed as a parameter to the DCDC_Init function.
- gDCDC_Event_McuV_OutputTargetChange_c: VDDMCU 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_McuV_OutputTargetVal_1_275_c = 0,
    gDCDC_McuV_OutputTargetVal_1_300_c,
    gDCDC_McuV_OutputTargetVal_1_325_c,
    gDCDC_McuV_OutputTargetVal_1_350_c,
    gDCDC_McuV_OutputTargetVal_1_375_c,
    gDCDC_McuV_OutputTargetVal_1_400_c,
    gDCDC_McuV_OutputTargetVal_1_425_c,
    gDCDC_McuV_OutputTargetVal_1_450_c,
    gDCDC_McuV_OutputTargetVal_1_475_c,
    gDCDC_McuV_OutputTargetVal_1_500_c,
    gDCDC_McuV_OutputTargetVal_1_525_c,
    gDCDC_McuV_OutputTargetVal_1_550_c,
    gDCDC_McuV_OutputTargetVal_1_575_c,
    gDCDC_McuV_OutputTargetVal_1_600_c,
    gDCDC_McuV_OutputTargetVal_1_625_c,
    gDCDC_McuV_OutputTargetVal_1_650_c,
    gDCDC_McuV_OutputTargetVal_1_675_c,
    gDCDC_McuV_OutputTargetVal_1_700_c,
    gDCDC_McuV_OutputTargetVal_1_725_c,
    gDCDC_McuV_OutputTargetVal_1_750_c,
    gDCDC_McuV_OutputTargetVal_1_775_c,
    gDCDC_McuV_OutputTargetVal_1_800_c
}dcdcMcuVOutputTargetVal_t;
```

Description:

Defines the output target values for the VDDMCU power line. In Buck mode this value is limited at 1.65 V (gDCDC_McuV_OutputTargetVal_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_075_c,
    gDCDC_1P8OutputTargetVal_2_100_c,
    gDCDC_1P8OutputTargetVal_2_150_c,
    gDCDC_1P8OutputTargetVal_2_200_c,
    gDCDC_1P8OutputTargetVal_2_250_c,
    gDCDC_1P8OutputTargetVal_2_300_c,
    gDCDC_1P8OutputTargetVal_2_350_c,
    gDCDC_1P8OutputTargetVal_2_400_c,
    gDCDC_1P8OutputTargetVal_2_450_c,
    gDCDC_1P8OutputTargetVal_2_500_c,
    gDCDC_1P8OutputTargetVal_2_550_c,
    gDCDC_1P8OutputTargetVal_2_600_c,
    gDCDC_1P8OutputTargetVal_2_650_c,
    gDCDC_1P8OutputTargetVal_2_700_c,
    gDCDC_1P8OutputTargetVal_2_750_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 1.8 V power line.

Location:

DCDC.h

Name:

```
typedef struct dcdcCalbackParam_tag
{
    dcdcEvent_t dcdcEvent;
    dcdcMcuVOutputTargetVal_t dcdcMcuVOutputTargetVal;
    dcdc1P8OutputTargetVal_t dcdc1P8OutputTargetVal;
    uint16_t dcdc1P8OutputMeasuredVal;
    uint16_t dcdcVbatMeasuredVal;
}dcdcCallbackParam_t;
```

Description:

Defines the parameter type the application callback provides to the user. The structure contains a notification event, the current output target values for the VDDMCU and 1.8 power lines, the measured value in mV of the 1.8 V 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 the application callback type. The application callback is passed to the DCDC driver in the DCDC configuration structure passed as a 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;
    dcdcMcuVOutputTargetVal_t dcdcMcuVOutputTargetVal;
```

```
    dcdc1P8OutputTargetVal_t dcdc1P8OutputTargetVal;
}dcdcConfig_t;
```

Description:

Defines the DCDC configuration structure type. A pointer to a DCDC configuration structure is passed as a 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) is monitored and an application callback is called, the application callback address, and the desired output target values for the VDDMCU and 1.8 V power lines.

Location:

DCDC.h

3.20.5 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, keeps a pointer to the structure internally. If the DCDC configuration structure mode in Bypass mode, the function takes no other action. 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 is a 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
(
dcdcMcuVOutputTargetVal_t dcdcMcuVOutputTargetVal,
dcdc1P8OutputTargetVal_t dcdc1P8OutputTargetVal
);
```

Description

The function enables changing the desired output target voltages at run-time. The values that are set in DCDC registers are the closest possible values to the desired values with respect to the current battery voltage value. The desired values are kept internally in the driver and are automatically set in the DCDC registers when the battery voltage value allows it. If this function isn’t called, the driver uses the output target values from the DCDC configuration structure.

Parameters

dcdcMcuVOutputTargetVal_t dcdcMcuVOutputTargetVal is the desired output target voltage level for the VDDMCU power line.

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

Returns

FALSE if a parameter is not valid or if this function is called prior to the DCDC_Init.

TRUE if executed successfully.

Prototype

```
uint16_t DCDC_McuVOutputTargetTomV(dcdcMcuVOutputTargetVal_t dcdcMcuVOutputTarget);
```

Description

The function converts dcdcMcuVOutputTargetVal_t to millivolts.

Parameters

dcdcMcuVOutputTargetVal_t dcdcMcuVOutputTarget is the dcdcMcuVOutputTargetVal_t constant to be converted to millivolts.

Returns

The conversion result as uint16_t.

Prototype

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

Description

The function converts the dcdc1P8OutputTargetVal_t to millivolts.

Parameters

dcdc1P8OutputTargetVal_t dcdc1P8OutputTarget is 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 the 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 either Boost mode or Buck mode.

Parameters

None

Returns

FALSE if this function is called prior to the DCDC_Init.

TRUE otherwise.

Prototype

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

Description

The function restores continuous mode settings in the DCDC hardware. In the Connectivity software system, the DCDC works in a 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 either Boost or Buck modes and DCDC_PrepareForPulsedMode was called before entering low-power.

Parameters

None

Returns

FALSE if this function is called prior to the 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 modes.

Parameters

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

Returns

FALSE if this function is called prior to the 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 the DCDC_Init.

TRUE if the variable value is set at the 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 the DCDC_Init or if the DCDC operating mode is not Buck mode, the system returns from this function. Otherwise the system, is 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 as follows:

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.

Note that the callback is called directly from the DCDC interrupt.

Parameters

pfDCDCPSwitchCallback_t pfPSwCallback is the callback to be called when the 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 the DCDC_Init.

TRUE otherwise.

3.21 Over-the-Air Programming Support

3.21.1 Overview

This module includes APIs for the over-the-air image upgrade process.

A Server device receives an image over the serial interface from a PC or other device thorough FSCI commands. If the Server has an image storage, the image is saved locally. If not, the image is requested chunk by chunk:

With image storage

- OTA_RegisterToFsci()
- OTA_InitExternalMemory()
- OTA_WriteExternalMemory()
- …
- OTA_WriteExternalMemory()

Without image storage:

- OTA_RegisterToFsci()
- OTA_QueryImageReq()
- OTA_ImageChunkReq()
- …
- OTA_ImageChunkReq()

A Client device processes the received image by computing the CRC and filter unused data and stores the received image into a non-volatile storage. After the entire image has been transferred and verified, the Client device informs the Bootloader application that a new image is available, and that the MCU must be reset to start the upgrade process. See the following command sequence:

- OTA_StartImage()
- OTA_PushImageChunk() and OTA_CrcCompute ()
- …
- OTA_PushImageChunk() and OTA_CrcCompute ()
- OTA_CommitImage()
- OTA_SetNewImageFlag()
- ResetMCU()

NOTE

For K32W061/QN9090/JN5189, with support for OTA on internal flash, a dedicated bootloader application is not required. In case of OTA on external flash a Secondary Stage Bootloader (SSBL) must be used.

3.21.2 Constant macro definitions

Name:

```
#define gEnableOTAServer_d (0)
```

Description:

Enables/disables the OTAP Server capabilities

Name:

```
#define gUpgradeImageOnCurrentDevice_d (0)
```

Description:

If enabled, the image received over the serial interface will be written into the MCU (for testing purpose).

Name:

```
#define gOtaVersion_c (0x01)
```

Description:

Defines the version of the Over-the-Air programming support module.

Name:

```
#define gOtaVerifyWrite_d (1)
```

Description:

If enabled, the OtaSupport module verifies that the image chunks are correctly written to flash.

Name:

```
#define gBootValueForTRUE_c (0x00)
```

Description:

Do not change. Internal use only.

Name:

```
#define gBootValueForFALSE_c (0xFF)
```

Description:

Do not change. Internal use only.

Name:

```
#define gBootData_StartMarker_Value_c
```

Description:

This represents the value of the start marker of the internal image storage. It is used by bootloader to determine the start address of the internal storage if the boot flags are invalid. If the value of this define is changed, then the define from the source code of the bootloader must also be changed to remain in sync.

NOTE

This flag is not used on K32W061/JN5189/QN9090.

Name:

```
#define gBootData_StartMarker_Offset_c
```

Description:

This represents the offset to the location of the start marker.

Name:

```
#define gBootData_Marker_Size_c
```

Description:

The represents size reserved for the start marker. If internal storage is not used, the size is zero.

NOTE

This flag is not used on K32W061/JN5189/QN9090.

Name:

```
#define gBootData_ImageLength_Offset_c (gBootData_StartMarker_Offset_c + gBootData_Marker_Size_c)
```

Description:

Defines the offset to the location of the image length field. Do not change. Internal use only.

NOTE

This flag is not used on K32W061/JN5189/QN9090.

Name:

```
#define gBootData_ImageLength_Size_c (0x04)
```

Description:

Defines the size of the image length field. Do not change. Internal use only.

NOTE

This flag is not used on K32W061/JN5189/QN9090.

Name:

```
#define gBootData_SectorsBitmap_Offset_c gEepromAlignAddr_d(gBootData_ImageLength_Offset_c + \
    gBootData_ImageLength_Size_c)
```

Description:

Defines the offset to the location of the sector bitmap field. Do not change. Internal use only.

NOTE

This flag is not used on K32W061/JN5189/QN9090.

Name:

```
#define gBootData_SectorsBitmap_Size_c (32)
```

Description:

Defines the length of the sector bitmap field. Do not change. Internal use only.

NOTE

This flag is not used on K32W061/JN5189/QN9090.

Name:

```
#define gBootData_Image_Offset_c gEepromAlignAddr_d(gBootData_SectorsBitmap_Offset_c + \
    gBootData_SectorsBitmap_Size_c)
```

Description:

Defines the offset to the location of the image binary. Do not change. Internal use only.

Name:

```
#define gFlashParams_MaxImageLength_c
```

Description:

Defines the maximum possible size for an image. Do not change. Internal use only.

Name:

```
#define gBootFlagUnprogrammed_c
```

Description:

This represents the default value of a boot flag. The value depends on the flash controller program unit size.

3.21.3 User-defined data type definitions

Name:

```
typedef enum {
    gOtaSucess_c = 0,
    gOtaNoImage_c,
    gOtaUpdated_c,
    gOtaError_c,
    gOtaCrcError_c,
    gOtaInvalidParam_c,
    gOtaInvalidOperation_c,
    gOtaExternalFlashError_c,
    gOtaInternalFlashError_c,
} otaResult_t;
```

Description:

Status codes

Name:

```
typedef enum {
    gUpgradeImageOnCurrentDevice_c = 0,
    gUseExternalMemoryForOtaUpdate_c,
    gDoNotUseExternalMemoryForOtaUpdate_c,
    gOtaTestingProcess
}otaMode_t;
```

Description:

Modes of operation for the OTAP Server.

Name:

```
typedef otaResult_t (*pfOTA_Callback_t)(uint8_t* pBuffer, uint16_t len);
```

Description:

Callback function used by OTAP Server application which notify the reception of a FSCI command.

Name:

```
typedef struct otaServer_AppCB_tag{
    pfOTA_Callback_t otaServerImgNotifyCnf;
    pfOTA_Callback_t otaServerSetModeCnf;
    pfOTA_Callback_t otaServerQueryImageCnf;
    pfOTA_Callback_t otaServerPushChunkCnf;
    pfOTA_Callback_t otaServerCancelImgCnf
    pfOTA_Callback_t otaServerAbortProcessCnf;
    pfOTA_Callback_t otaServerSetFileVersPoliciesCnf;
}otaServer_AppCB_t;
```

Description:

Callback functions used by the OTAP Server to notify the reception of a certain FSCI command.

Name:

```
typedef PACKED_STRUCT
{
    uint8_t  newBootImageAvailable[FSL_FEATURE_FLASH_PFLASH_BLOCK_WRITE_UNIT_SIZE];
    uint8_t  bootProcessCompleted[FSL_FEATURE_FLASH_PFLASH_BLOCK_WRITE_UNIT_SIZE];
    uint8_t  version[FSL_FEATURE_FLASH_PFLASH_BLOCK_WRITE_UNIT_SIZE];
    uint8_t  internalStorageAddr [FSL_FEATURE_FLASH_PFLASH_BLOCK_WRITE_UNIT_SIZE];

}bootInfo_t;
```

Description:

This is the structure of the boot flags. Do not change this structure.

3.21.4 API primitives

OTA_RegisterToFsci

Prototype:

```
otaResult_t OTA_RegisterToFsci
(
uint32_t fsciInterface,
```

```
otaServer_AppCB_t *pCB
);
```

Description: Register the OTAP Server callback functions to an FSCI instance.

Parameters:

Name	Type	Direction	Description
fsciInterface	uint32_t	[IN]	Id of the FSCI interface
pCB	otaServer_AppCB_t*	[IN]	Pointer to a structure containing callback functions

Returns:

The error code.

OTA_StartImage

Prototype:

```
otaResult_t OTA_StartImage
(
uint32_t length
);
```

Description: Start the Over-the-Air image upgrade process (Client).

Parameters:

Name	Type	Direction	Description
length	uint32_t	[IN]	Length of the image to be received

Returns:

The error code.

OTA_PushImageChunk

Prototype:

```
otaResult_t OTA_PushImageChunk
(
uint8_t* pData,
uint16_t length,
uint32_t* pImageLength,
uint32_t *pImageOffset
);
```

Description: Saves image chunks received over-the-air into a non-volatile storage (Client).

Parameters:

Name	Type	Direction	Description
pData	uint8_t*	[IN]	Pointer to the image chunk

Table continues on the next page...

Table continued from the previous page...

Name	Type	Direction	Description
length	uint16_t	[IN]	Length in bytes of the image chunk.
pImageLength	uint32_t*	[OUT]	Pointer to a location where to store the currently written bytes. Can be NULL if not used.
pImageOffset	uint32_t*	[IN]	If it is not null contains the current offset of the image

Returns:

The error code.

OTA_CommitImage

Prototype:

```
otaResult_t OTA_CommitImage
(
uint8_t* pBitmap
);
```

Description: Writes the sector bitmap and length into the image storage (Client). If the internal storage is used, the start marker is also written.

NOTE

This argument is not used on K32W061/JN5189/QN9090

Parameters:

Name	Type	Direction	Description
pBitmap	Uint8_t*	[IN]	Pointer to the flash sector bitmap

Returns:

The error code.

OTA_CancelImage

Prototype:

```
void OTA_CancelImage
(
Void
);
```

Description: Aborts the image upgrade process (Client).

Returns:

None.

OTA_SetNewImageFlag

Prototype:

```
void OTA_SetNewImageFlag
(
Void
);
```

Description: Sets flags to inform the Bootloader that a new image is available for upgrade (Client).

Returns:

None.

OTA_InitExternalMemory

Prototype:

```
otaResult_t OTA_InitExternalMemory
(
Void
);
```

Description: Initializes the non-volatile image storage.

Returns:

The error code.

OTA_ReadExternalMemory

Prototype:

```
otaResult_t OTA_ReadExternalMemory
(
uint8_t* pData,
uint16_t length,
uint32_t address
);
```

Description: Reads data from the image storage.

Parameters:

Name	Type	Direction	Description
pData	Uint8_t*	[OUT]	Pointer where data from the image storage will be saved
length	Uint16_t	[IN]	Data length in bytes
address	uint32_t	[IN]	Offset into the image storage

Returns:

The error code.

OTA_WriteExternalMemory

Prototype:

```
otaResult_t OTA_WriteExternalMemory
(
uint8_t* pData,
uint16_t length,
```

```
uint32_t address
);
```

Description: Writes data into the image storage.

Parameters:

Name	Type	Direction	Description
pData	Uint8_t*	[IN]	Pointer to data to be written into the image storage
length	Uint16_t	[IN]	Data length in bytes
address	uint32_t	[IN]	Offset into the image storage

Returns:

The error code.

OTA_CrcCompute

Prototype:

```
uint16_t OTA_CrcCompute
(
uint8_t *pData,
uint16_t lenData,
uint16_t crcValueOld
);
```

Description: Computes the CRC over the specified image data.

Parameters:

Name	Type	Direction	Description
pData	Uint8_t*	[IN]	Pointer to the data
lenData	Uint16_t	[IN]	Data length in bytes
crcValueOld	Uint16_t	[IN]	Current CRC value

Returns:

New CRC value.

OTA_EraseExternalMemory

Prototype:

```
otaResult_t OTA_EraseExternalMemory
(
Void
);
```

Description: Erases the entire image storage.

Returns:

The error code.

OTA_EraseBlock

Prototype:

```
otaResult_t OTA_EraseBlock
(
uint32_t address
);
```

Description: Erases a sector/block of the image storage.

Parameters:

Name	Type	Direction	Description
address	uint32_t	[IN]	Address of the sector/block to be erased

Returns:

The error code.

OTA_ImageChunkReq

Prototype:

```
void OTA_ImageChunkReq
(
uint32_t offset,
uint16_t len,
uint16_t devId
);
```

Description: Requests a certain image chunk over the FSCI interface (Server).

Parameters:

Name	Type	Direction	Description
offset	uint32_t	[IN]	Offset of the image
len	Uint16_t	[IN]	Length of the chunk
devId	Uint16_t	[IN]	The Id of the device

Returns:

None.

OTA_QueryImageReq

Prototype:

```
void OTA_QueryImageReq
(
uint16_t devId,
uint16_t manufacturer,
uint16_t imgType,
uint32_t fileVersion
);
```

Description: Requests an image with the specified parameters over the FSCI interface (Server).

Parameters:

Name	Type	Direction	Description
devId	Uint16_t	[IN]	Id of the device
manufacturer	Uint16_t	[IN]	The manufacturer code
imgType	Uint16_t	[IN]	The type of the image
fileVersion	uint32_t	[IN]	Current image version

Returns:

None.

3.22 Bootloader

NOTE

This section does not apply to K32W061/JN5189/QN9090. Instead, a Secondary Stage Bootloader (SSBL) can be used for these platforms.

3.22.1 Overview

The Bootloader is a program which resides in a reserved area of the flash memory of the device. It starts before the application, checks if an application image update needs to be performed, and, if true, it proceeds to replace the current application image with a new image. If an update does not need to be performed, the bootloader terminates and the application starts to run.

There are two types of bootloaders available in the Connectivity Software package:

- OTAP Bootloader
- FSCI Bootloader

The figure below shows the memory layout of the device with the relevant sections and their size, such as the bootloader, the application, and the reserved areas.

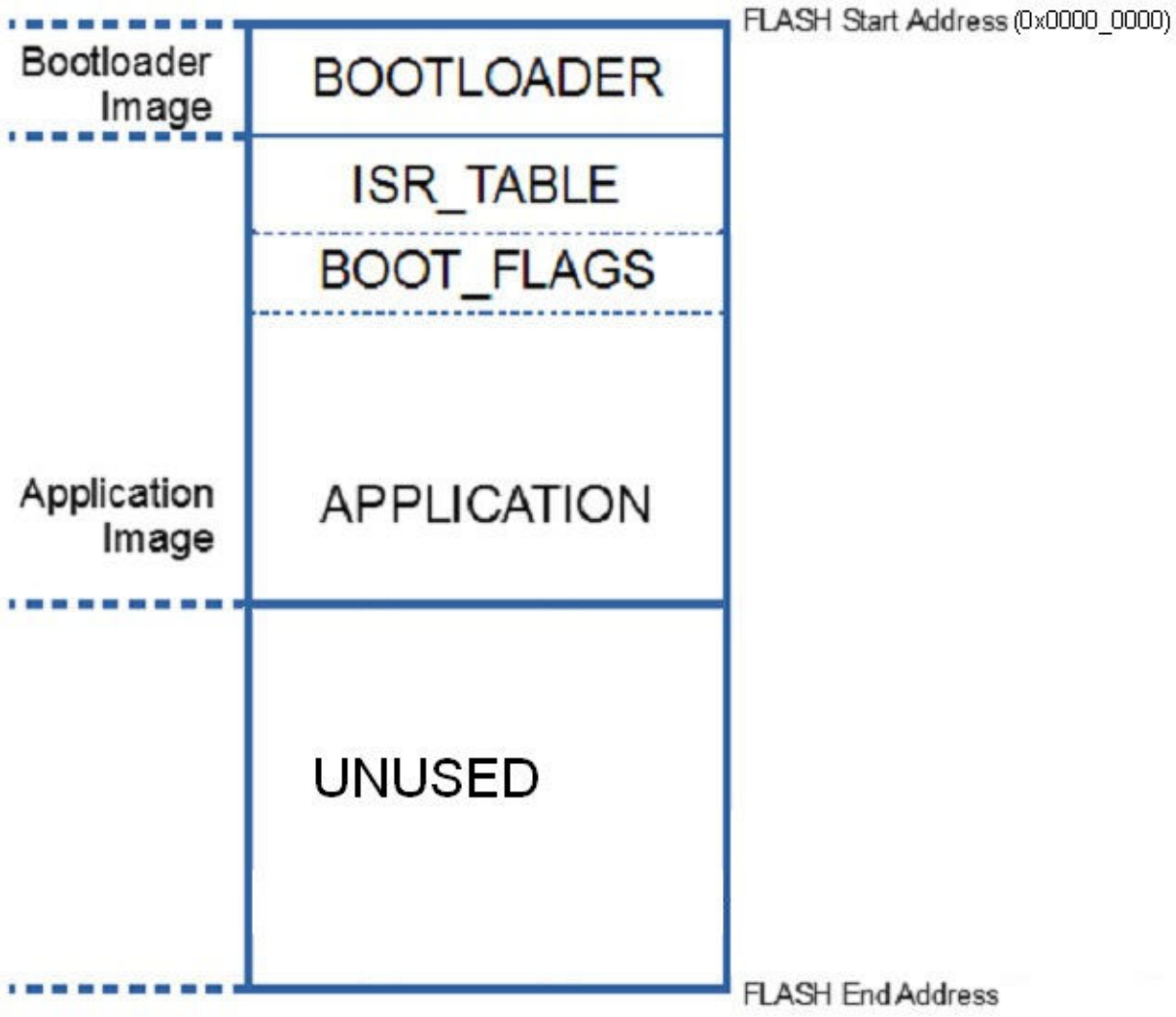


Figure 18. Bootloader overview

The application image is divided into the following sections:

1. The ISR_TABLE section contains the MCU interrupt table which has a fixed reserved size. This section must be placed immediately after the bootloader image.
2. The BOOT_FLAGS section contains bootloader flags, target bootloader version, and address of the internal storage.

   The OTAP Bootloader looks for this section immediately after the ISR_TABLE section which has fixed size.

   - New Image Flag – set by the application to notify the OTAP Bootloader that a new image is available. This flag is set by calling the *OTA_SetNewImageFlag()* function from the *OtaSupport* module.
   - Image Upgrade Complete Flag – set by the OTAP Bootloader when the new image copy process is completed successfully.
   - Bootloader Version – bootloader version expected by the application; set at compile time.
   - Internal Storage start address – set by the application to inform the OAP Bootloader the flash address of the internal storage at which the new image is available. If this flag is invalid, the external image storage will be used.
3. The APPLICATION section contains the application code, read only data, and, optionally , an internal image storage area and an application NVM section:
4. The optional application NVM section is placed at the highest flash address range available (not reserved), if present.

5. The optional internal image storage area is placed before the NVM section if the NVM is present, or at the highest available flash address if the NVM is not present.

If the application uses the default connectivity linker file, the user can add the *gUseBootloaderLink_d=1*symbol to the project linker configuration to offset the application's binary and accommodate the Bootloader.

NOTE

The bootloader can only update the application's binary code. The bootloader image itself is not updated.

3.22.2 OTAP Bootloader

This bootloader is used in over-the-air software update process. The application must store the newly received image into a non-volatile memory. The bootloader code reads it from the location and replaces the existing application.

At startup, the bootloader checks dedicated new image flags in non-volatile memory. If the flags are set, it proceeds to replace the current running application image with a new image received over-the-air. The new image can be retrieved from external or internal storage depending on the configuration and available memory resources of the device. After the bootloader copies the new image, it programs the "Image update complete" flag, then resets the MCU.

If the new image flags are not set, the OTAP Bootloader gives control of the MCU to the current application.

If the image upgrade progress is interrupted before it is finished, by a power loss for example, the bootloader restarts the copy procedure on the next MCU reset. It uses flags in non-volatile memory, which are set only when the image copy process has been completed successfully, to restart the copy procedure.

The OTAP Bootloader expects a certain image format in the image storage location which is identical regardless of internal or external storage.

The format of the raw image is described in this table:

Field Name	Field Length (bytes)	Description
Image Size	4	This is the *Binary Image* field size. It is set by the *OTA_CommitImage()* function from the *OtaSupport* module. Its value is equal to the sum of all image parts written using the *OTA_PushImageChunk()* function.
Sector Bitmap	32	This field contains a sector bitmap of the flash memory of the target device which tells the bootloader which sectors to overwrite and which to leave intact. This field is the *Value* field of the Sector Bitmap Sub-element of the image file sent over-the-air. This field is set by the *OTA_CommitImage()* function from the *OtaSupport* module. The format of this field is least significant byte first and least significant bit first for each byte with the least significant bytes and bits standing for the lowest memory sections of the flash.

Table continues on the next page...

Table continued from the previous page...

Field Name	Field Length (bytes)	Description
		The OTAP Bootloader is configured not to overwrite itself regardless of the sector bitmap settings of the flash area it resides in. This setting can be changed in the OTAP Bootloader application.
Binary Image	variable	This field contains the binary application which is written to the APPLICATION section of the flash. This field is the *Value* field of the Upgrade Image Sub-element of the image file sent over-the-air. This field is gradually set by each call to the *OTA_PushImageChunk()* function.

NOTE

The OTAP Bootloader does not overwrite the last flash sector which contains the hardware-dependent data.

3.22.3 FSCI Bootloader

The FSCI Bootloader uses a serial interface (UART or SPI Slave) to load a new binary image into the MCU's flash thorough a set of FSCI commands. The flash section containing the Bootloader code is not altered.

The FSCI Bootloader can be triggered in these ways:

- Write a flag in the BOOT_FLAGS section flash memory and reset the MCU (software approach)
- Assert a GPIO pin and reset the MCU (hardware approach).

To trigger the Bootloader from software, the application must program the first word of the BOOT_FLAGS section to 0xF5C18007 value.

NOTE

As in the case of the previous bootloaders, the application linker file must be modified to use the flash memory located after the one reserved for the bootloader.

If the bootloader is triggered using the software, then if the MCU resets or goes through a power cycle, the bootloader does not give control to the application unless a new image is updated successfully. If the bootloader is triggered by an external source (GPIO) and the MCU resets during the upgrade process, the external trigger must be asserted for the bootloader to remain in upgrade mode and not jump to the application. The Host MCU must restart the FSCI bootloader after the image upgrade process ended successfully.

The table below shows the default settings for the Bootloader GPIO trigger pin.

- Bootloader GPIO trigger pin settings

Board	GPIO Trigger Pin	Description
FRDM-KW40Z	PORTA18	Board Switch 4

Table continues on the next page...

Table continued from the previous page...

Board	GPIO Trigger Pin	Description
FRDM-KW41Z	PORTC5	Board Switch 4
FRDM-KW24D	PORTE4	Board Switch 1
TWR-KW24D512	PORTC4	Board Switch 1
TWR-KW21D256	PORTC4	Board Switch 1
QN908XCDK	GPIO_A 19	Board Switch 2
FRDM-KW36	PORTC2	Board Switch 3

NOTE

Because the Bootloader application uses the default clock configuration Kinetis MCU-based platforms, (FLL clocked from the slow internal reference clock), the UART baud rate may not be accurate. To resolve this issue, use an external crystal oscillator as a reference clock, or use the UART automatic baud rate calibration.

Depending on the available flash memory for the bootloader code both SPI and UART interfaces maybe enabled. In this case, the first interface on which the bootloader receives data is used to load the new application image. If only one serial interface is used, the other interface can be disabled by changing the UART or SPI-specific macro definition.

If OTAP bootloader functionality is required with the FSCI bootloader, the ***gUseOTAPBootloader_d*** macro must be set to 1 and the corresponding source files has to be added into the project. The OTAP and FSCI bootloaders can coexist depending on available flash memory in bootloader section.

3.22.3.1 Bootloader FSCI configuration

The following macros can be changed to configure the FSCI protocol.

gFsciIncluded_c

Description: Enable/disable the FSCI module

```
#ifndef gFsciIncluded_c
#define gFsciIncluded_c 0
#endif
```

gFsciMaxPayloadLen_c

Description: The maximum size in bytes of a FSCI command.

```
#ifndef gFsciMaxPayloadLen_c
#define gFsciMaxPayloadLen_c 2060
#endif
```

gFsciLenHas2Bytes_c

Description: Number of bytes representing the payload length.

```
#ifndef gFsciLenHas2Bytes_c
#define gFsciLenHas2Bytes_c 1 /* Boolean */
#endif
```

gFsciBootUseSeqNo_c

Description: Use sequence numbers for image chunks. If enabled, the first payload byte of the *FSCI-PushImageChunk.Request* represents the current sequence number.

```
#ifndef gFsciBootUseSeqNo_c
#define gFsciBootUseSeqNo_c 1
#endif
```

gFsciUseCRC_c

Description: If enabled, the *FSCI-CommitImage.Request* must contain the CRC of the image. This CRC is compared with the CRC computed on the written image. If the received CRC does not match the computed CRC, an error status is reported by the *FSCI-CommitImage.Response*.

```
#ifndef gFsciUseCRC_c
#define gFsciUseCRC_c 1
#endif
```

gFsciTxAck_c

Description: If enabled, the bootloader sends an ACK frame for every FSCI command received correctly.

```
#ifndef gFsciTxAck_c
#define gFsciTxAck_c 0
#endif
```

gFsciRxAck_c

Description: If enabled, the bootloader will wait for an ACK frame for every transmitted FSCI command.

```
#ifndef gFsciRxAck_c
#define gFsciRxAck_c 0
#endif
```

mFsciRxAckTimeoutMs_c

Description: The amount of time in milliseconds that the bootloader should wait for an ACK frame.

```
#define mFsciRxAckTimeoutMs_c 100
```

mFsciTxRetryCnt_c

Description: The number of retransmissions for a transmitted FSCI command for which an ACK frame was not received.

```
#define mFsciTxRetryCnt_c 4
```

gBoot_UseSpiSlave_d

Description: Enable/disable the SPI Slave interface used by the FSCI Bootloader.

```
#ifndef gBoot_UseSpiSlave_d
#define gBoot_UseSpiSlave_d 1
#endif
```

gSpi_DummyChar_d

Description: The byte to be sent to the SPI Master when the Slave has no data to send.

```
#ifndef gSpi_DummyChar_d
#define gSpi_DummyChar_d (0xFF)
#endif
```

gSPISlaveDapTxLogicOne_d

Description: Enable if the Data Available Pin should signal the SPI Master on logic 1.

```
#ifndef gSPISlaveDapTxLogicOne_c
#define gSPISlaveDapTxLogicOne_c (0)
#endif
```

gSPISlave_RxBufferSize_d

Description: The Rx buffer size of the SPI Slave driver.

```
#ifndef gSpiSlave_RxBufferSize_d
#define gSpiSlave_RxBufferSize_d (2100)
#endif
```

gBoot_UseUart_d

Description: Enable/disable the SPI Slave interface used by the FSCI Bootloader.

```
#ifndef gBoot_UseUart_d
#define gBoot_UseUart_d 1
#endif
```

gBoot_UseUartCalibration_d

Description: This macro can be found in the “Uart.h” file. It enables the UART automatic baud rate calibration protocol (FC protocol from AN2295).

```
#ifndef gBoot_UseUartCalibration_d
#define gBoot_UseUartCalibration_d 0
#endif
```

3.22.3.2 Bootloader HW configuration

The following defines can be found in the MCU-specific configuration file, MKxxxxx_cfg.h, and can be modified to change the GPIO trigger pin, UART interface, or SPI interface.

The macro definitions from below represent the default hardware configuration:

GPIO trigger pin

```
#define BOOT_PIN_ENABLE_SIM_SCG_REG SIM_SCGC5
#define BOOT_PIN_ENABLE_SIM_SCG_MASK SIM_SCGC5_PORTC_MASK
#define BOOT_PIN_ENABLE_PORT_BASE PORTC_BASE_PTR
#define BOOT_PIN_ENABLE_GPIO_BASE PTC_BASE_PTR
#define BOOT_PIN_ENABLE_NUM 4
```

UART Configuration

```
#define BOOT_UART_IRQ INT_UART1_RX_TX
#define BOOT_UART_BASE UART1_BASE_PTR
#define BOOT_UART_SIM_SCG_REG SIM_SCGC4
#define BOOT_UART_SIM_SCG_MASK SIM_SCGC4_UART1_MASK
#define BOOT_UART_BAUD_RATE 57600
#define BOOT_UART_GPIO_PORT_RX PORTE_BASE_PTR
#define BOOT_UART_GPIO_PORT_TX PORTE_BASE_PTR
#define BOOT_UART_GPIO_PORT_SIM_SCG_REG SIM_SCGC5
#define BOOT_UART_GPIO_PORT_SIM_SCG_MASK SIM_SCGC5_PORTE_MASK
#define BOOT_PIN_UART_ALTERNATIVE 3
```

```
#define BOOT_UART_GPIO_PIN_RX 1
#define BOOT_UART_GPIO_PIN_TX 0
```

SPI Configuration

```
#define BOOT_SPI_Slave_IRQ INT_SPI0
#define BOOT_SPI_Slave_BaseAddr SPI0_BASE_PTR
#define BOOT_SPI_Slave_PORT_SIM_SCG SIM_SCGC5
#define cSPI_Slave_PORT_SIM_SCG_Config_c SIM_SCGC5_PORTC_MASK
#define BOOT_SPI_Slave_SIM_SCG SIM_SCGC6
#define cSPI_Slave_SIM_SCG_Config_c SIM_SCGC6_SPI0_MASK
#define BOOT_SPI_Slave_SCLK_PCR PORTC_PCR5
#define BOOT_SPI_Slave_MOSI_PCR PORTC_PCR6
#define BOOT_SPI_Slave_MISO_PCR PORTC_PCR7
#define BOOT_SPI_Slave_SSEL_PCR PORTC_PCR4
#define BOOT_SPI_Slave_ALTERNATIVE 2
```

3.22.3.3 Bootloader Flash Security

Flash Protection

By default, the flash section used by the Bootloader is not secured. To protect this section, the macro value from below must be changed to 0xFFFFFFFE.

```
#ifndef gFlashProtection_c
#define gFlashProtection_c 0xFFFFFFFF /* Flash is not write protected */
#endif
```

Flash Backdoor Key

If the application needs to access the bootloader flash section, a “Back Door Key” must be set in the bootloader code. The application must use this key to unsecure the flash until the next MCU reset.

```
#ifndef gFlashBackDoorKey1_d
#define gFlashBackDoorKey1_d 0xFFFFFFFF
#endif

#ifndef gFlashBackDoorKey2_d
#define gFlashBackDoorKey2_d 0xFFFFFFFF
#endif
```

3.22.3.4 Supported FSCI commands

FSCI-CPUReset.Request

Direction: Host to Bootloader

Description: This command resets the MCU. No confirmation is sent.

Parameter	Size (bytes)	Comments
OpGroup	1	0xA3
OpCode	1	0x08
Length	2/1	0x00 – This request does not have any payload.

FSCI-Error.Indication

Direction: Bootloader to Host

Description: Indicates that the last received command caused an error.

Parameter	Size (bytes)	Comments
OpGroup	1	0xA3
OpCode	1	0xFE
Length	2/1	0x01
Error code	1	0xF7 – Unknown OpGroup 0xFD – Unknown OpCode

FSCI-StartImage.Request

Direction: Host to Bootloader

Description: Start the image upgrade process. If the upgrade process is already started, an error code is returned in the following confirmation.

Parameter	Size (bytes)	Comments
OpGroup	1	0xA3
OpCode	1	0x29
Length	2/1	Payload size
Image Length	4	Number of bytes to be downloaded.

FSCI-StartImage.Confirm

Direction: Bootloader to Host

Description: Returns the status of the FSCI-StartImage.Request.

Parameter	Size (bytes)	Comments
OpGroup	1	0xA4
OpCode	1	0x29
Length	2/1	Payload length
Status	1	0x00 – Success 0x05 – Invalid parameter 0x06 – Invalid Operation
Version	1	Protocol version implemented in the Bootloader
External Memory Supported	1	0x00 – External memory not available 0x01 – External memory support available

FSCI-CancelImage.Request

Direction: Host to Bootloader

Description: Terminates the image upgrade process.

Parameter	Size (bytes)	Comments
OpGroup	1	0xA3
OpCode	1	0x2C
Length	2/1	0x00

FSCI-CancelImage.Confirm

Direction: Bootloader to Host

Description: Confirms that the image upgrade process has been terminated.

Parameter	Size (bytes)	Comments
OpGroup	1	0xA4
OpCode	1	0x2C
Length	2/1	0x01
status	1	0x00 - Success

FSCI-SetMode.Request

Direction: Host to Bootloader

Description: Sets the upgrade mode. Reserved for feature use.

Parameter	Size (bytes)	Comments
OpGroup	1	0xA3
OpCode	1	0x28
Length	2/1	Payload length
Upgrade mode	1	Reserved

FSCI-SetMode.Confirm

Direction: Bootloader to Host

Description: Always returns success.

Parameter	Size (bytes)	Comments
OpGroup	1	0xA4
OpCode	1	0x28
Length	2/1	0x01
Status	1	0x00 - Success

FSCI-PushImageChunk.Request

Direction: Host to Bootloader

Description: Sends the next image chunk to be written to flash. Image chunks must be consecutive.

If the *gFsciBootUseSeqNo_c* is enabled, the first byte of the payload, represents the *SeqNo*.

Parameter	Size (bytes)	Comments
OpGroup	1	0xA3
OpCode	1	0x2A
Length	2/1	Payload length
SeqNo	1/0	Optional, can be disabled from SW. The sequence number of the current image chunk.
Image Chunk	variable	Binary image data

FSCI-PushImageChunk.Confirm

Direction: Bootloader to Host

Description: Status of the flash programming operation.

Parameter	Size (bytes)	Comments
OpGroup	1	0xA4
OpCode	1	0x2A
Length	2/1	0x01
Status	1	0x00 – Success 0x03 – Unexpected SeqNo 0x06 – Invalid Operation 0x08 – MCU flash error

FSCI-CommitImage.Request

Direction: Host to Bootloader

Description: Marks the upgrade process completed. The CRC field is ignored if the *gFsciUseCRC_c* macro is not enabled.

Parameter	Size (bytes)	Comments
OpGroup	1	0xA3
OpCode	1	0x2B
Length	2/1	0x00
CRC	0/2	The CRC of the new image (Optional)

FSCI-CommitImage.Confirm

Direction: Bootloader to Host

Description: Returns the status of the FSCI-CommitImage.Request. The Host must reset the MCU after this confirmation is received to run the new application.

Parameter	Size (bytes)	Comments
OpGroup	1	0xA4
OpCode	1	0x2B
Length	2/1	0x01
Status	1	0x00 – Success 0x04 – CRC mismatch 0x06 – Invalid Operation 0x08 – MCU flash error

3.22.3.5 Image Update Sequence Chart

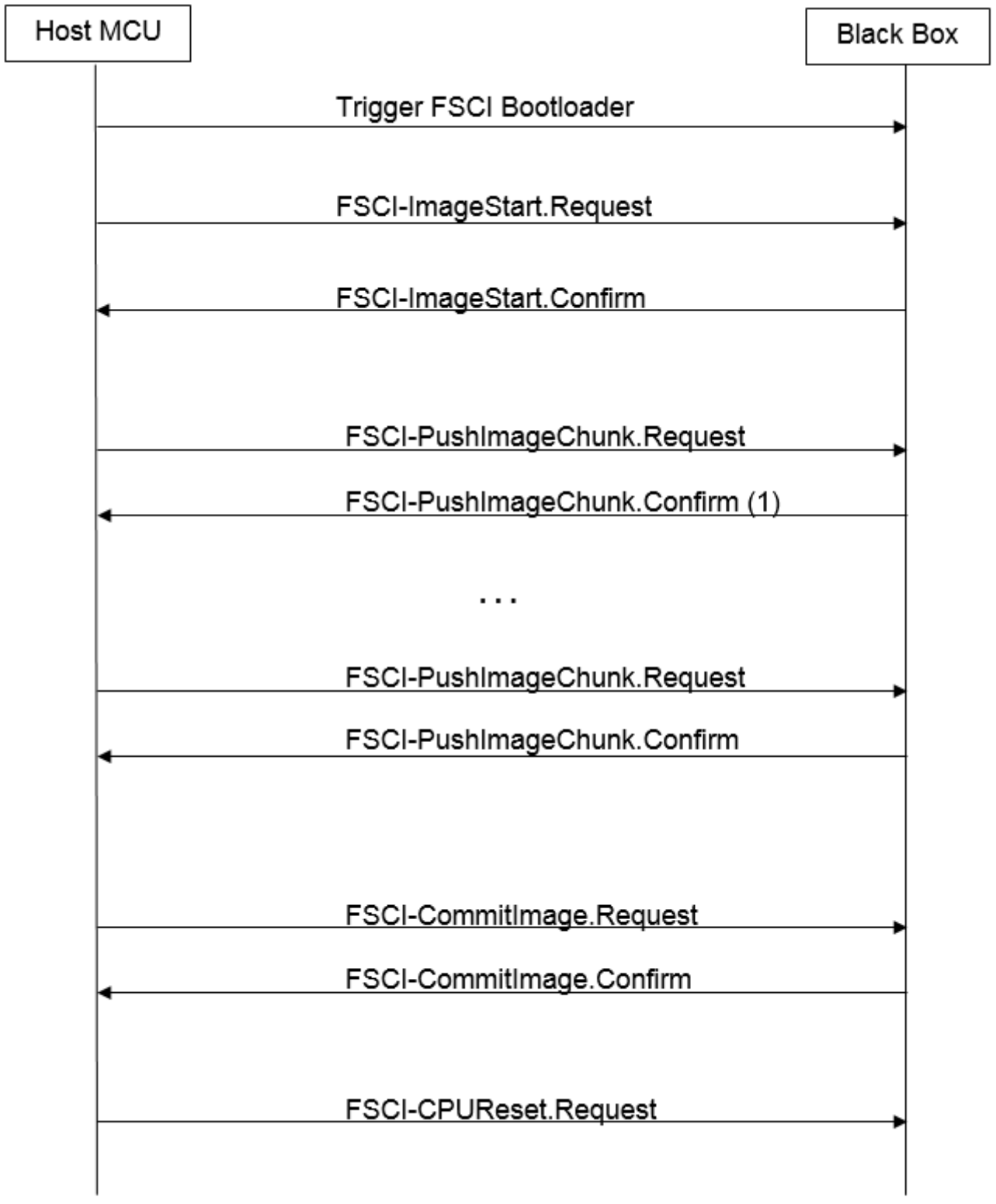


Figure 19. Image Update Sequence Chart

3.22.3.6 Image Update Chart when FSCI ACKs are enabled

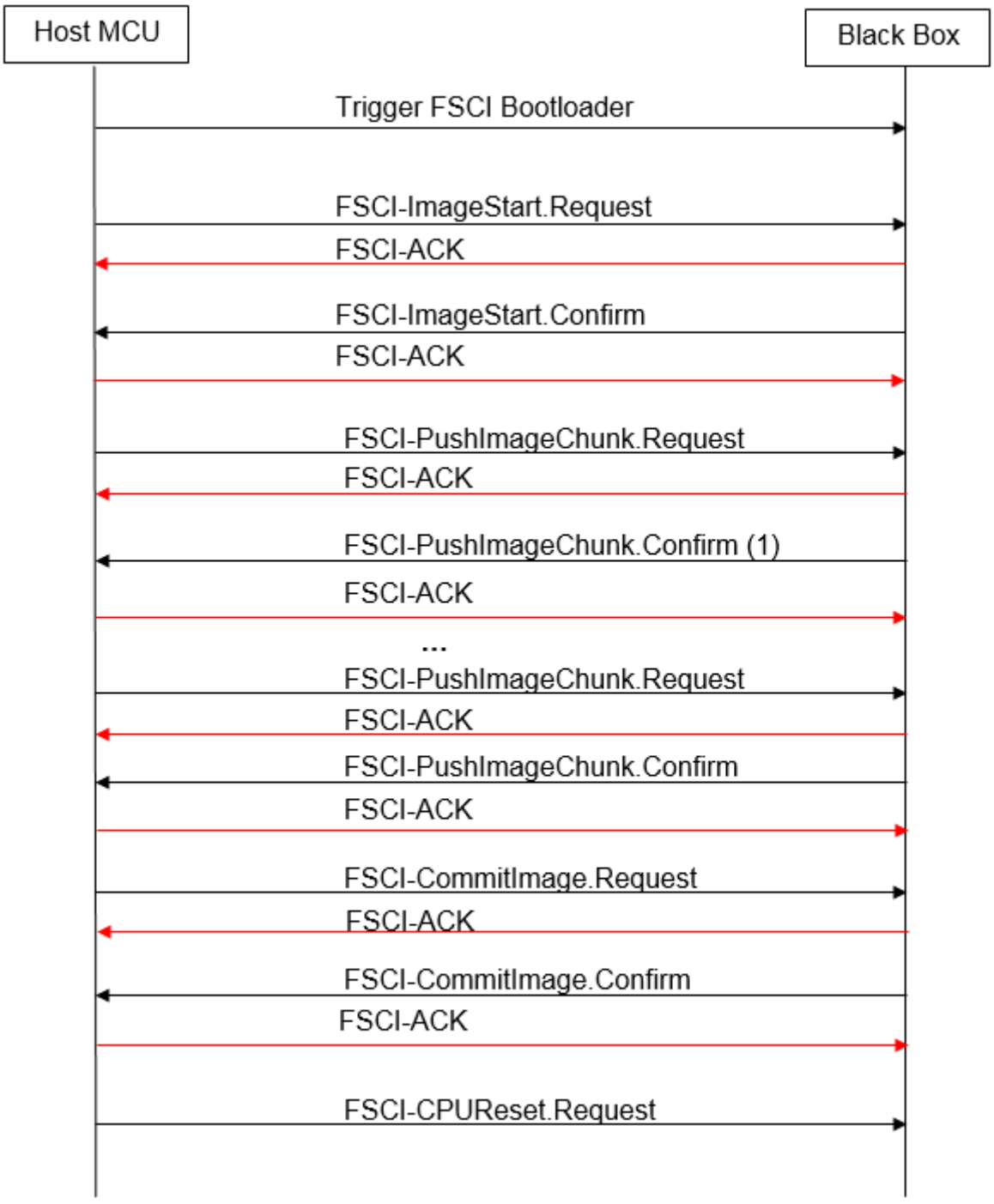


Figure 20. Image Update Chart when FSCI ACKs are enabled

3.22.3.7 Application Integration Example

The first step to integrate the FSCI bootloader with an application involves the linker file considerations described in the “Overview” section and are common for all the bootloader variants.

The second step pertains to the interaction of the application and the bootloader through the BOOT_FLAGS sector.

For a regular Connectivity application, any flash write/erase function implementation is suitable. The recommended API is the FlashAdapter from the Kinetis Connectivity Framework, found in the *<SDK_Path>\middleware\wireless\framework\Flash\Internal* \ folder and distributed in Kinetis connectivity software packages. The main functions are:

```
uint32_tNV_FlashEraseSector(
    uint32_t dest,
    uint32_t size
    );

uint32_tNV_FlashProgram(
    uint32_t dest,
    uint32_t size,
    uint8_t* pData
    );
```

For applications of the FSCI Black Box type, the command to write/erase the respective bytes needs to be invoked from a host processor through an FSCI remote procedure call. For this purpose, a dedicated FSCI command was implemented:

```
mFsciEnableFsciBootloaderReq_c = 0xCF,
```

Chapter 4
Revision History

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

Table 8. Revision history

Revision	Date	Substantive changes
0	06/2015	Initial release.
1	09/2015	Added KW40Z Low-Power, DCDC and production data storage.
2	02/2016	Updated Section 3.12 , “ Security Library ”. Added Section 3.16 , “ Mobile Wireless Systems Coexistence ” and Section 3.21 , “ Bootloader ”.
3	04/2016	Added support for KW4x.
4	07/2016	Updated for Kinetis W.
7	12/2017	Updated for QN9080x
8	03/2018	Updated for KW36. Updated Section 3.14 , “ Timers Manager ", Section 3.14 , “ Serial manager ", and Section 3.14 , “ Low-power library "
9	09/2018	Updated for QN9080x
10	12/2018	Added Memory Manager debug feature description. Updated the bootloader and Over-the-Air programming chapters
11	06/2019	Updated for KW35
12	10/2019	Updated for QN908x
13	06/2020	Updated for KW35 Maintenance Release4

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

Information in this document is provided solely to enable system and software implementers to use NXP products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. NXP reserves the right to make changes without further notice to any products herein.

NXP makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does NXP assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in NXP data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including “typicals,” must be validated for each customer application by customer's technical experts. NXP does not convey any license under its patent rights nor the rights of others. NXP sells products pursuant to standard terms and conditions of sale, which can be found at the following address: nxp.com/SalesTermsandConditions.

While NXP has implemented advanced security features, all products may be subject to unidentified vulnerabilities. Customers are responsible for the design and operation of their applications and products to reduce the effect of these vulnerabilities on customer’s applications and products, and NXP accepts no liability for any vulnerability that is discovered. Customers should implement appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP, the NXP logo, NXP SECURE CONNECTIONS FOR A SMARTER WORLD, COOLFLUX, EMBRACE, GREENCHIP, HITAG, I2C BUS, ICODE, JCOP, LIFE VIBES, MIFARE, MIFARE CLASSIC, MIFARE DESFire, MIFARE PLUS, MIFARE FLEX, MANTIS, MIFARE ULTRALIGHT, MIFARE4MOBILE, MIGLO, NTAG, ROADLINK, SMARTLX, SMARTMX, STARPLUG, TOPFET, TRENCHMOS, UCODE, Freescale, the Freescale logo, AltiVec, CodeWarrior, ColdFire, ColdFire+, the Energy Efficient Solutions logo, Kinetis, Layerscape, MagniV, mobileGT, PEG, PowerQUICC, Processor Expert, QorIQ, QorIQ Qonverge, SafeAssure, the SafeAssure logo, StarCore, Symphony, VortiQa, Vybrid, Airfast, BeeKit, BeeStack, CoreNet, Flexis, MXC, Platform in a Package, QUICC Engine, Tower, TurboLink, EdgeScale, EdgeLock, eIQ, and Immersive3D are trademarks of NXP B.V. All other product or service names are the property of their respective owners. AMBA, Arm, Arm7, Arm7TDMI, Arm9, Arm11, Artisan, big.LITTLE, Cordio, CoreLink, CoreSight, Cortex, DesignStart, DynamIQ, Jazelle, Keil, Mali, Mbed, Mbed Enabled, NEON, POP, RealView, SecurCore, Socrates, Thumb, TrustZone, ULINK, ULINK2, ULINK-ME, ULINK-PLUS, ULINKpro, µVision, Versatile are trademarks or registered trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. The related technology may be protected by any or all of patents, copyrights, designs and trade secrets. All rights reserved. Oracle and Java are registered trademarks of Oracle and/or its affiliates. The Power Architecture and Power.org word marks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org.

For more information, please visit: http://www.nxp.com
For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 01 June 2020
Document identifier: CONNFWRM

arm