NXP Semiconductors
User's Guide

Document Number: 802154MPADG

Rev. 4, 09/2016

IEEE 802.15.4 MAC/PHY

Application Developer's Guide

1. Introduction

This user's guide describes the IEEE 802.15.4™ Multiple PAN MAC and PHY layers implementation for Kinetis platforms, with Real-Time Operating System (RTOS) support through the Connectivity Framework. The 802.15.4 MAC/PHY software is designed for use with the following families of low-rate, low-power, 2.4 GHz and sub-1 GHz Industrial, Scientific, and Medical (ISM) band transceivers:

- MKW2x System-in-Package based on the Kinetis ARM® Cortex®-M4 32-bit MCU.
- Split solution based on the Kinetis ARM Cortex-M4/M0+ 32-bit MCU and MCR20A transceiver (referred to as MCR20).
- MKW01 System-in-Package based on the Kinetis ARM Cortex-M0+ 32-bit MCU.
- KW4x System-on-Chip based on the Kinetis ARM Cortex-M0+ 32-bit MCU.

Throughout this user's guide, the term transceiver refers to the digital and analog radio peripheral inside the platforms mentioned above.

Contents

2. Scope and Objective

This is a user's guide for the Kinetis-based 802.15.4 MACPHY library. It describes building an IEEE 802.15.4-based application or network layer over RTOS, supported through the Freescale Connectivity Framework.

2.1. Audience

This document is primarily intended for 802.15.4 MAC application developers.

2.2. Organization

The core of the document consists of these sections:

- Section 3, "IEEE 802.15.4 MAC/PHY Software Overview"—this section describes the Freescale 802.15.4 MAC/PHY software build environment, source file structure, and hardware setup.
- Section 4, "MAC/Network Layer Interface Description"—this section describes the MAC/PHY interface.
- Section 5, "Interfacing to the 802.15.4 MAC software"—this section describes interfacing an application to the MAC and using the MAC interface functions.
- Section 6, "Feature Descriptions"—this section describes the Freescale 802.15.4 MAC/PHY software features, with focus on implementing specific details of the 802.15.4 standard.
- Section 7, "Interfacing to the 802.15.4 MAC software"—this section describes the NWK layer accessing a MAC black box through a serial interface.

2.3. Conventions

This document uses the following notational conventions:

- `Courier monospaced type` indicates commands, command parameters, code examples, expressions, data types, and directives.
- *Italic type* indicates replaceable command parameters.

All source code examples are in C.

2.4. Definitions, acronyms, and abbreviations

This table defines the abbreviations used throughout this document.

Table 1. Definitions, acronyms, and abbreviations

Term/acronym	Definition
ACK	Acknowledgement Frame
API	Application Programming Interface
ASP	Application Support Package
APP	Application
CAP	Contention Access Period
CFP	Contention Free Period
FFD	Full Function Device (as specified in the 802.15.4 Standard)
FFDNGTS	FFD without GTS support
FFDNB	FFD without beacon support
FFDNBNS	FFD without beacon or security support
GPIO	General-Purpose Input Output
GTS	Guaranteed Time Slot
HW	Hardware
IRQ	Interrupt Request
ISR	Interrupt Service Routine
MAC	Medium Access Control
MCPS	MAC Common Part Sublayer- Service Access Point
MCU	Microcontroller
MLME	MAC Sublayer Management Entity
MSDU	MAC Service Data Unit
NWK	Network Layer
PAN	Personal Area Network
PAN ID	PAN Identification
PCB	Printed Circuit Board
PHY	Physical Layer

Table 1. Definitions, acronyms, and abbreviations

Term/acronym	Definition
PIB	PAN Information Base
PSDU	PHY Service Data Unit
RF	Radio Frequencies
RFD	Reduced Function Device (as specified in the 802.15.4 Standard)
RFDNB	RFD without beacon support
RFDNBNS	RFD without beacon or security support
SAP	Service Access Point
SW	Software

2.5. References

Table 2. References

Title	Reference	Version	Location
IEEE Standard for Information technology - Local and metropolitan area networks - Specific requirements - Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low Rate Wireless Personal Area Networks (WPANs)	IEEE Std 802.15.4-2006	2006	ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=11161
IEEE Standard for Local and metropolitan area networks - Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs)	IEEE Std 802.15.4-2011 (Revision of IEEE Std 802.15.4-2006)	2011	ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6012485
IEEE Standard for Local and metropolitan area networks Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs) Amendment 1: MAC sublayer	IEEE Std 802.15.4e-2012 (Amendment to IEEE Std 802.15.4-2011)	2012	ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6185525
IEEE Standard for Local and metropolitan area networks Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs) Amendment 3: Physical Layer (PHY) Specifications for Low-Data-Rate, Wireless, Smart Metering Utility Networks	IEEE Std 802.15.4g-2012 (Amendment to IEEE Std 802.15.4-2011)	2012	ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6190698

3. IEEE 802.15.4 MAC/PHY Software Overview

This section provides an overview of the IEEE 802.15.4 Standard background and describes Freescale 802.15.4 MAC/PHY implementation, parametric details, build environment, source file structure, and hardware setup.

NOTE

Become familiar with the IEEE Std 802.15.4-2003, Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs) and/or IEEE Std 802.15.4™-2006, Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs) as required. This document does not describe all the differences between the 802.15.4 Standard revisions from 2003, 2006, or 2011, except for those relevant to the 802.15.4 MAC/PHY software.

Freescale 802.15.4 MAC/PHY software is intended for the following platforms:

- MKW2x 2.4 GHz Platform-in-Package based on Kinetis 32-bit (ARM Cortex-M4) MCU
- Split solution based on Kinetis ARM Cortex-M4/M0+ 32-bit MCU and the MCR20A transceiver
- MKW01 sub-1 GHz Platform-in-Package based on Kinetis 32-bit (ARM Cortex-M0+) MCU
- MKW4x 2.4GHz System-on-Chip based on Kinetis 32-bit (ARM Cortex-M0+) MCU

This guide supports Kinetis 32-bit platforms and 802.15.4 Standards (2003, 2006, and 2011).

- MAC 2011 is available on all Kinetis platforms
- MAC 2006 is available on all Kinetis platforms
- MAC 2003 is available on all Kinetis platforms
- Differences in use and services and differences in the standards deployed in the software are highlighted throughout the guide as necessary

3.1. Understanding the 802.15.4 Standard

The 802.15.4 Standard is developed for Wireless Personal Area Networks (WPANs). WPANS convey information over short distances between the participants in the network. They enable implementing of small, power-efficient, inexpensive solutions for a wide range of applications and device types. The key characteristics of the 802.15.4 Standard network are:

- Over-the-air data rate of 250 kbit/s in the 2.4 GHz ISM band
- 16 independent communication channels in the 2.4 GHz band
- Large networks (up to 65534 devices)
- Devices use Carrier Sense Multiple Access with Collision Avoidance (CSMA-CA) to access the medium
- Devices use Energy Detection (ED) for channel selection (implemented in the SCAN primitive)
- Devices inform the application about the quality of the wireless link—Link Quality Indication (LQI) is reported as a part of the Data Indication primitive

The 802.15.4 Standard defines two network topologies; both topologies use only one central device (the PAN coordinator). The PAN coordinator is the principal controller of the network.

- Star Network Topology—in a star network, all communication in the network is directed either to or from the PAN coordinator. Communication between non-PAN coordinator devices is not possible.
- Peer-to-Peer Network Topology—in a peer-to-peer network, communication occurs between any two devices in the network, as long as they are within each other's range.

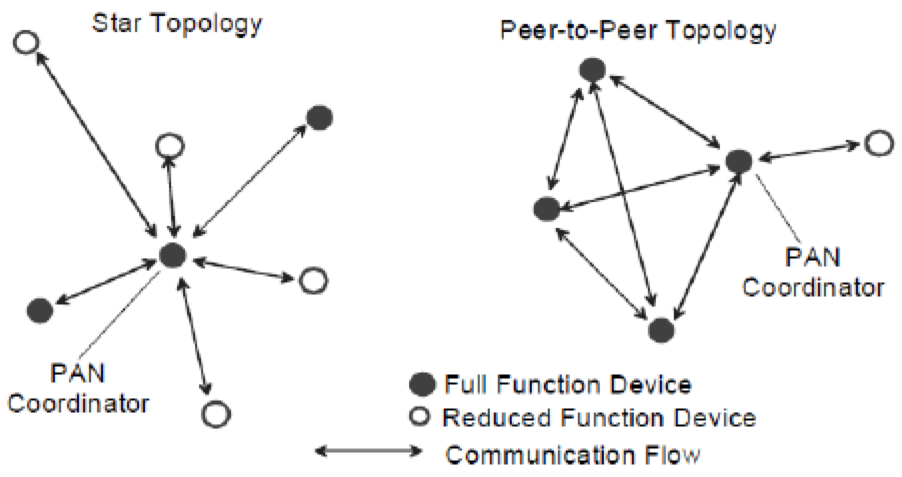

Figure 1. Star and peer-to-peer network

If a device wants to join an 802.15.4 network, it must associate with a device that is already part of the network. This enables other devices to associate with it. Multiple devices can associate with the same device (as shown in Figure 2). A device that has other devices associated with it is a coordinator to those devices. The coordinator provides synchronization services to the devices that are associated with it through the transmission of beacon frames (as shown in Figure 2). In a star network, there is only one PAN coordinator. In a peer-to-peer network, there can be multiple coordinators plus the PAN coordinator.

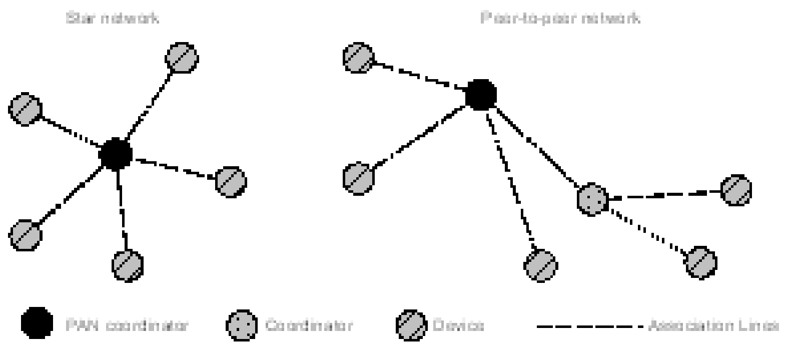

Figure 2. Peer-to-peer and star networks (PAN coordinator)

Networks (both star and peer-to-peer) operate either in the beacon mode or the non-beacon mode. In the beacon mode, all coordinators within the network transmit synchronization frames (beacon frames) to their associated devices. All data transmissions between the coordinator and its associated devices occur in the active period following the beacon frame, as shown in this figure.

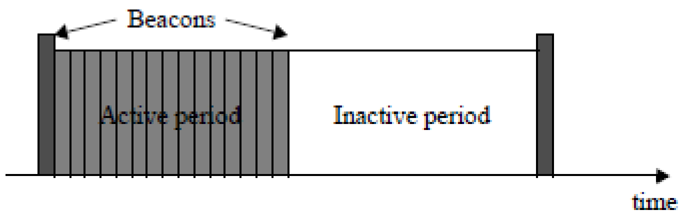

Figure 3. Beacon frame timing

In both the non-beacon and beacon networks, the application transmits data in these ways:

- Direct Data Transfer—transfer of data from the device to the coordinator using direct data transfer takes place as soon as the channel is free. For beacon networks, direct data is transferred during the active period.
- Indirect Data Transfer—transfer of data from a coordinator to a device using indirect data transfer is stored in the coordinator's queue and transferred to the device when the latter makes a poll request.
- Beaconed Tree Mode—a peer-to-peer network operating in the beacon mode experiences beacon collision, which can result in the possible loss of synchronization. The ZigBee® 1.0 specification outlines the Beaconed Tree Mode, which is a synchronized peer-to-peer network topology. The advantage of a Beaconed Tree Mode network is lower power requirements. A Beaconed Tree Mode network node is active for a short duration (the active portion of the superframe), and it enters low-power mode (sleep) during inactive periods of the superframe. Freescale 802.15.4 software supports Beaconed Tree Mode, as described in *Freescale 802.15.4 Media Access Controller (MAC) Demo Applications User's Guide* (document 802154MPDAUG).

3.2. 802.15.4 Standard evolution (2003, 2006, 2011, e-2012, g-2012)

This section lists some of the additions to the 802.15.4 Standard for 2006 when compared to the 802.15.4 Standard for 2003, as implemented in the MAC software. See the appropriate 802.15.4 Standard specification for further details.

- 2006 PHY enhancements:
  - Added a channel page to allow more flexibility for new channel allocations
  - Simplified transceiver states (removed Busy_Rx and Busy_Tx)
  - Modified and added PHY PIB attributes
    - Modified phyChannelsSupported attribute
    - Supports PHY PIB access through the MAC SAP
    - Added phyCurrentPage attribute—the current PHY channel page
    - Added phyMaxFrameDuration attribute—the maximum number of symbols in a frame
    - Added phySHRDuration attribute—the duration of the synchronization header (SHR) in symbols
    - Added phySymbolsPerOctet attribute—the number of symbols per octet

- 2006 MAC Enhancements:
  - Reduced complexity, reduced MAC overhead, and resolved long association times
  - Improved security
  - Supports more detailed beacon scheduling
  - Supports distributed shared (beacon) timebase
  - Supports multicast by employing broadcast frame transmission procedures
  - Provided new CCM suite that consolidates CTR and CBC-MAC suites
  - Removed the Access Control List (ACL)
  - Appended the Auxiliary Security Header (ASH) to the addressing field as part of the MHR
  - Redesigned the MAC security PIB attribute table
  - Clarified security operations and optimized storage of keying material
  - Improved data authenticity, replay protection, and simplified protection parameter setup
  - A single key can now be used for different protection levels in a frame
  - Allows unsecured communications until a higher layer sets up the key
- 2011 PHY Enhancements:
  - In 2007, two new PHYs were added as an amendment, one of which supported accurate ranging. MAC capability to support ranging was added as a part of this ammendment.
  - In 2009, two new PHY amendments were approved; one to provide operation in frequency bands specific to China, and the other for operation in frequency bands specific to Japan.
    - CSS PHY—Chirp Spread Spectrum (CSS) employing Differential Quadrature Phase-Shift Keying (DQPSK) modulation, operating in the 2450 MHz band.
    - UWB PHY—combines Burst Position Modulation (BPM) and BPSK modulation, operating in the sub-gigahertz and 3-10 GHz bands.
    - MPSK PHY—M-ary Phase-Shift Keying (MPSK) modulation, operating in the 780 MHz band.
    - GFSK PHY—Gaussian Frequency-Shift Keying (GFSK), operating in the 950 MHz band.
- 2011 MAC Enhancements:
  - Ranging procedure
  - Aloha—media access mechanism
  - New security suite
- e-2012 Enhancements:
  - Enhanced Beacon
  - Enhanced Beacon Request
  - Information Elements
  - Fast Association Procedure
  - Low Energy

    - Coordinated Sampled Listening
    - Receiver Initiated Transmission
  - AMCA—Asynchronous Multi-channel Adaptation
  - LLDN—Low-Latency Deterministic Network
    - simplified CSMA-CA
    - new API primitives
    - LLDN Superframe structure
  - TSCH—Time-Slotted Channel Hopping
    - CCA algorithm
    - CSMA-CA algorithm
    - new API primitives
    - TSCH Slot frame structure
  - DSME—Deterministic and Synchronous multichannel Extension
- g-2012 Enhancements:
  - This ammendment specifies alternate PHYs in addition to those of IEEE Std. 802.15.4-2011:
    - Multirate and Multiregional Frequency Shift Keying (MR-FSK) PHY
    - Multirate and Multiregional Orthogonal Frequency Division Multiplexing (MR-OFDM) PHY
    - Multirate and Multi-Regional Offset Quadrature Phase-Shift Keying (MR-O-QPSK) PHY
  - The SUN PHYs support multiple data rates in bands ranging from 169 MHz to 2450 MHz

3.3. System overview

The following figure shows a block diagram of the system. The application uses the lower layers to implement a wireless application based on the Freescale 802.15.4 software.

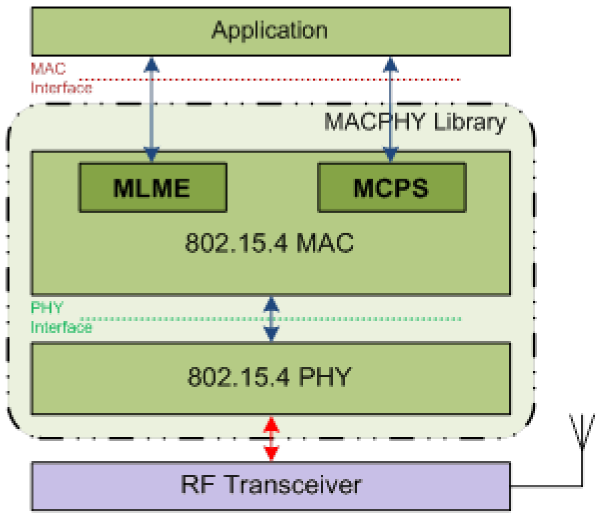

Figure 4. System block diagram

The application can be anything, and it is entirely up to you. Some examples are:

- Dedicated MAC application
- ZigBee network layer
- Proprietary stack

The layer below the application (as shown in Figure 4) is the 802.15.4 MAC (or just MAC). The MAC provides these interfaces to the application:

- MLME (MAC Sublayer Management Entity) Interface—this interface is used for all 802.15.4 MAC commands. For example, the application uses this interface to send the MLME-ASSOCIATE.Request primitive, and it also receives the MLME-ASSOCIATE.Confirm primitive on this interface. This interface is defined in the 802.15.4 Standard.
- MCPS (MAC Common Part Sublayer) Interface—this interface is used for all 802.15.4 data-related primitives. The application uses this interface to send and receive data. This interface is defined in the 802.15.4 Standard.

As shown in Figure 4, the two layers at the bottom are the PHY and the actual radio (including hardware driver). The application cannot access the PHY and hardware layers directly.

NOTE

The application uses the MAC interfaces to implement the desired functionality. Section 4,”MAC/Network Layer Interface Description”, describes the interfaces in complete detail.

3.4. 802.15.4 MAC/PHY parametric information

The following subsections describe the main parametric information for the Freescale 802.15.4 MAC/PHY software.

3.4.1. Clocking and timing information

3.4.1.1. MKW2x

The clock requirements described here are for the Kinetis Cortex-M4-based platform. The CPU clock and bus clock are running at 48 MHz. To ensure that the timing constraints are set as required by the 802.15.4 Standard, the 2.4 GHz digital radio die is clocked from a 32 MHz external oscillator. The digital radio die then supplies a 4 MHz clock signal to the MCU's PLL.

3.4.1.2. MCR20

Configure the MCU to not exceed the maximum SPI speed when communicating with the MCR20 transceiver (16 MHz for write access and 9 MHz for read access).

3.4.1.3. MKW01

The clock requirements described here are for the Kinetis Cortex-M0+-based platform. The CPU clock runs at 48 MHz, and the bus clock runs at 24 MHz. To ensure that the timing constraints are set as required by the 802.15.4 Standard, the sub-1 GHz digital radio die is clocked from various types of external oscillators, depending on the frequency band used. The digital radio die then supplies an identical clock signal to the MCU. The most common clocking frequencies are 32 MHz and 30 MHz.

3.4.1.4. MKW4x

The CPU clock runs at 32 MHz, and the bus clock runs at 16 MHz. To ensure that the timing constraints are set as required by the 802.15.4 Standard, the 2.4 GHz digital radio die is clocked from a 32-MHz external oscillator.

NOTE

All application tasks must have lower priority than the MAC tasks.

3.5. 802.15.4 MAC/PHY software build environment

Freescale 802.15.4 MAC/PHY software for MKW2x, MKW01, MKW4x, and MCR20-based platforms is built using IAR Embedded Workbench IDE.

3.5.1. MAC libraries

These are the available MAC libraries:

- 802.15.4_mac_06begts_cm0_iar.a
- 802.15.4_mac_06begts_cm4_iar.a
- 802.15.4_mac_06eleg_cm0_iar.a
- 802.15.4_mac_06etschg_cm0_iar.a
- 802.15.4_mac_06g_cm0_iar.a
- 802.15.4_mac_06rfd_cm0_iar.a
- 802.15.4_mac_06rfd_cm4_iar.a
- 802.15.4_mac_06_cm0_iar.a
- 802.15.4_mac_06_cm4_iar.a
- 802.15.4_mac_11eleg_cm0_iar.a
- 802.15.4_mac_11etschg_cm0_iar.a
- 802.15.4_mac_11g_cm0_iar.a
- 802.15.4_mac_11_cm0_iar.a
- 802.15.4_mac_11_cm4_iar.a
- 802.15.4_mac_thrrfd_cm0_iar.a
- 802.15.4_mac_thrrfd_cm4_iar.a
- 802.15.4_mac_thr_cm0_iar.a
- 802.15.4_mac_thr_cm4_iar.a

Table 3. MAC taxonomy

Abreviation	Description
06	2006 security
11	2011 security
g	MAC used with g PHY
e	802.14.5e features (LE/TSCH/DSME)
LE	4e Low Energy features (CSL, RIT)
BE	Beacon Enabled
GTS	Guaranteed Time Slots
TSCH	4e Time-Slotted Channel Hopping
DSME	4e Deterministic and Synchronous Multichannel Extension
ZP	ZigBeePro-specific implementations
M0	MAC compiled for Cortex-M0+
M4	MAC compiled for Cortex-M4
RFD	Reduced Function Device
THR	Thread-specific implementation
THRRFD	Thread RFD-specific implementation

3.5.2. Adding user applications to the build environment

Freescale 802.15.4 MAC/PHY software includes the Freescale 802.15.4 MAC libraries, the Freescale 802.15.4 (2.4 GHz and sub-1 GHz) PHY source code, and IAR project files (`.ewp`) only.

- You can add your application directly on top of the build environment, but doing that requires both in-depth knowledge of the 802.15.4 Standard and wireless application experience.
- Freescale strongly recommends that you base your application development on *Freescale 802.15.4 MAC/PHY My_Wireless_App_demo* application example software. This software is described in *Freescale 802.15.4 Media Access Controller (MAC) Demo Applications User's Guide* (document 802154MPDAUG). Generate *MyWirelessApp* source code using Freescale components.
- For more information, see Freescale ZigBee home page at www.nxp.com/zigbee.

4. MAC/Network Layer Interface Description

This section describes the MAC/PHY interface for FDD, RFD, and their derivatives.

4.1. General MAC/network interface information

The interface between the Network Layer (NWK) and the two services that the MAC sublayer provides (MAC data service (MCPS) and the MAC Logical Management Entity (MLME)) is based on service primitives passed from one layer to the other through a layer Service Access Point (SAP). The following SAPs must be implemented as functions in the application:

1. `typedef resultType_t (*MLME_NWK_SapHandler_t) (nwkMessage_t* pMsg, instanceId_t upperInstanceId);`
   The `MLME_NWK_SapHandler_t` type function passes primitives from the MLME to the NWK.
2. `typedef resultType_t (*MCPS_NWK_SapHandler_t) (mcpsToNwkMessage_t* pMsg, instanceId_t upperInstanceId);`
   The `MCPS_NWK_SapHandler_t` type function passes primitives from the MCPS to the NWK.

Two SAP handlers are likewise implemented in the MAC. They accept messages in the opposite direction from the NWK to the MLME, and MCPS.

1. `resultType_t NWK_MLME_SapHandler( mlmeMessage_t* pMsg, instanceId_t macInstanceId );`
   The `NWK_MLME_SapHandler` function passes primitives from the NWK to the MLME.
2. `resultType_t NWK_MCPS_SapHandler(nwkToMcpsMessage_t* pMsg, instanceId_t macInstanceId);`
   The `NWK_MCPS_SapHandler` function passes primitives from the NWK to the MCPS.

NOTE

If the Multiple Pan feature is used, then the SAP handlers are discriminated by the `xxxInstanceId` parameter.

Call the SAP handler functions directly. The MLME and MCPS service primitives use different types of messages, as defined in the *MacInterface.h* interface header file. Internally, the `NWK_MLME` and `NWK_MCPS` SAP handler functions can place a message into the message queue. To process queued messages, the MAC task must run. Because the NWK and MLME/MCPS interfaces are based on messages being passed to a few SAPs, each message must have an identifier. These identifiers are shown in the following four tables. Some of the identifiers are not supported for some of the device types.
For example, the *MLME-GTS.Request* primitive is available for the *gMacFeatureSet_06M4_d* feature set, but the functionality is not supported.

The following table lists all message identifiers in the MLME to NWK direction. They cover all MLME confirm and indication primitives:

Table 4. Primitives in the MLME to NWK direction

Message identifier (macMessageId_t)	802.15.4 MLME to NWK primitives
gMlmeAssociateInd_c	MLME-ASSOCIATE.Indication
gMlmeAssociateCnf_c	MLME-ASSOCIATE.Confirm
gMlmeDisassociateInd_c	MLME-DISASSOCIATE.Indication
gMlmeDisassociateCnf_c	MLME-DISASSOCIATE.Confirm
gMlmeBeaconNotifyInd_c	MLME-BEACON-NOTIFY.Indication
gMlmeGetCnf_c	MLME-GET.Confirm
gMlmeGtsInd_c	MLME-GTS.Indication
gMlmeGtsCnf_c	MLME-GTS.Confirm
gMlmeOrphanInd_c	MLME-ORPHAN.Indication
gMlmeResetCnf_c	MLME-RESET.Confirm
gMlmeRxEnableCnf_c	MLME-RX-ENABLE.Confirm
gMlmeScanCnf_c	MLME-SCAN.Confirm
gMlmeCommStatusInd_c	MLME-COMM-STATUS.Indication
gMlmeSetCnf_c	MLME-SET.Confirm
gMlmeStartCnf_c	MLME-START.Confirm
gMlmeSyncLossInd_c	MLME-SYNC-LOSS.Indication
gMlmePollCnf_c	MLME-POLL.Confirm
gMlmePollNotifyIndication_c	Freescale proprietary Poll Notify Indication
gMlmeSetSlotframeCnf_c	MLME-SET-SLOTFRAME.Confirm
gMlmeSetLinkCnf_c	MLME-SET-LINK.Confirm

Table 4. Primitives in the MLME to NWK direction

Message identifier (macMessageId_t)	802.15.4 MLME to NWK primitives
gMlmeTschModeCnf_c	MLME-TSCH-MODE.Confirm
gMlmeKeepAliveCnf_c	MLME-KEEP-ALIVE.Confirm
gMlmeBeaconCnf_c	MLME-BEACON.Confirm

The following table lists all message identifiers in the MCPS to NWK direction. They cover all MCPS confirm and indication primitives.

Table 5. Primitives in the MCPS to NWK direction

Message identifier (macMessageId_t)	802.15.4 MCPS to NWK primitives
gMcpsDataCnf_c	MCPS-DATA.Confirm
gMcpsDataInd_c	MCPS-DATA.Indication
gMcpsPurgeCnf_c	MCPS-PURGE.Confirm

The following table lists all the message identifiers in the NWK to the MLME direction. They cover all MLME request and response primitives.

Table 6. Primitives in the NWK to MLME direction

Message identifier (macMessageId_t)	802.15.4 NWK to MLME primitives
gMlmeAssociateReq_c	MLME-ASSOCIATE.Request
gMlmeAssociateRes_c	MLME-ASSOCIATE.Response
gMlmeDisassociateReq_c	MLME-DISASSOCIATE.Request
gMlmeGetReq_c	MLME-GET.Request
gMlmeGtsReq_c	MLME-GTS.Request
gMlmeOrphanRes_c	MLME-ORPHAN.Response
gMlmeResetReq_c	MLME-RESET.Request
gMlmeRxEnableReq_c	MLME-RX-ENABLE.Request

Table 6. Primitives in the NWK to MLME direction

Message identifier (macMessageId_t)	802.15.4 NWK to MLME primitives
gMlmeScanReq_c	MLME-SCAN.Request
gMlmeSetReq_c	MLME-SET.Request
gMlmeStartReq_c	MLME-START.Request
gMlmeSyncReq_c	MLME-SYNC.Request
gMlmePollReq_c	MLME-POLL.Request
gMlmeSetSlotframeReq_c	MLME-SET-SLOTFRAME.Request
gMlmeSetLinkReq_c	MLME-SET-LINK.Request
gMlmeTschModeReq_c	MLME-TSCH-MODE.Request
gMlmeKeepAliveReq_c	MLME-KEEP-ALIVE.Request
gMlmeBeaconReq_c	MLME-BEACON.Request

The following table lists all message identifiers in the NWK to the MCPS direction. They cover all MCPS request and response primitives.

Table 7. Primitives in the NWK to MCPS direction

Message identifier (macMessageId_t)	802.15.4 NWK to MCPS primitives
gMcpsDataReq_c	MCPS-DATA.Request
gMcpsPurgeReq_c	MCPS-PURGE.Request

4.2. Data types

This section describes the main C structures and data types used by the MAC/NWK interface.
The following data types represent the parameters needed to implement the services exposed by the SAP primitives. The data types assume that the endianess of the processor architecture with respect to the network-MAC interface is not in the scope of the upper layers. Instead, the MAC and/or lower layers are responsible for managing endianess of multibyte fields, such as the MAC extended address. As an example, the extended address is *uint64_t* type instead of *uint8_t[8]*. The pointer type is the exception from the little endian notation. The pointer type can be aligned to a suitable boundary and have the endianess of the CPU in question. Values for the various structure elements are defined by the 802.15.4 Standard. For example, Address Mode can take on the values 0 (No), 2 (Short), and 3 (Extended).

The structures described in Section 6.1.2.1, “Reset request” through Section, “6.15.4.3 “GTS-indication” are collected in single message type as unions, plus a message type that corresponds to the enumeration of the primitives. These are the structures that transport messages across the interface.

For messages from the MLME to the NWK, use this structure/union:

```
// MLME to NWK message
typedef STRUCT nwkMessage_tag
{
    macMessageId_t      msgType;
    UNION
    {
        mlmeAssociateInd_t      associateInd;
        mlmeAssociateCnf_t      associateCnf;
        mlmeDisassociateInd_t   disassociateInd;
        mlmeDisassociateCnf_t   disassociateCnf;
        mlmeBeaconNotifyInd_t   beaconNotifyInd;
        mlmeGetCnf_t            getCnf;
        mlmeGtsInd_t            gtsInd;
        mlmeGtsCnf_t            gtsCnf;
        mlmeOrphanInd_t         orphanInd;
        mlmeResetCnf_t          resetCnf;
        mlmeRxEnableCnf_t       rxEnableCnf;
        mlmeScanCnf_t           scanCnf;
        mlmeCommStatusInd_t     commStatusInd;
        mlmeSetCnf_t            setCnf;
        mlmeStartCnf_t          startCnf;
        mlmeSyncLossInd_t       syncLossInd;
        mlmePollCnf_t           pollCnf;
        mlmePollNotifyInd_t     pollNotifyInd;
        mlmeSetSlotframeCnf_t   setSlotframeCnf;
        mlmeSetLinkCnf_t        setLinkCnf;
        mlmeTschModeCnf_t       tschModeCnf;
        mlmeKeepAliveCnf_t      keepAliveCnf;
        mlmeBeaconCnf_t         beaconCnf;
    } msgData;
} nwkMessage_t;
```

For messages from the MCPS to the NWK, use this structure/union:

```
// MCPS to NWK message
typedef STRUCT mcpsToNwkMessage_tag
{
    macMessageId_t      msgType;
    UNION
    {
        mcpsDataCnf_t       dataCnf;
        mcpsDataInd_t       dataInd;
        mcpsPurgeCnf_t      purgeCnf;
        mcpsPromInd_t       promInd;
    } msgData;
} mcpsToNwkMessage_t;
```

Use the following structure/union for messages that are sent from the NWK to the MLME. Allocate the MLME message using *MEM_BufferAlloc(sizeof(mlmeMessage_t))*. The function returns a pointer to a memory location with sufficient number of bytes, or NULL when the memory pools are exhausted. Handle the NULL pointer in the same way as a confirm message with a status code of TRANSACTION_OVERFLOW. The allocated message that is sent to the MLME is freed automatically. Pay attention to the comments regarding allocation for the Set, Get, and Reset requests described in Section 6.1.2, "Configuration primitives."

```
// NWK to MLME message
typedef STRUCT mlmeMessage_tag
{
    macMessageId_t      msgType;
    UNION
    {
        mlmeAssociateReq_t      associateReq;
        mlmeAssociateRes_t      associateRes;
        mlmeDisassociateReq_t   disassociateReq;
        mlmeGetReq_t            getReq;
        mlmeGtsReq_t            gtsReq;
        mlmeOrphanRes_t         orphanRes;
        mlmeResetReq_t          resetReq;
        mlmeRxEnableReq_t       rxEnableReq;
        mlmeScanReq_t           scanReq;
        mlmeSetReq_t            setReq;
        mlmeStartReq_t          startReq;
        mlmeSyncReq_t           syncReq;
        mlmePollReq_t           pollReq;
        mlmeSetSlotframeReq_t   setSlotframeReq;
        mlmeSetLinkReq_t        setLinkReq;
        mlmeTschModeReq_t       tschModeReq;
        mlmeKeepAliveReq_t      keepAliveReq;
        mlmeBeaconReq_t         beaconReq;
    } msgData;
} mlmeMessage_t;
```

Use the following structure/union for messages that must be sent from the NWK to the MCPS. Allocate the *MCPS-DATA.Request* and *MCPS-PURGE.Request* using *MEM_BufferAlloc(sizeof(nwkToMcpsMessage_t))*. Allocate the *MCPS-DATA.Request* message using *MEM_BufferAlloc(sizeof(nwkToMcpsMessage_t) + gMaxPHYPacketSize_c)*. Both allocation functions return a pointer to a memory location with sufficient number of bytes, or NULL when the memory pools are exhausted. Handle the NULL pointer in the same way as a confirm message with a status code of TRANSACTION_OVERFLOW.

The allocated message that is sent to the MCPS is freed automatically.

```
// NWK to MCPS message
typedef STRUCT nwkToMcpsMessage_tag
{
    macMessageId_t      msgType;
    UNION
    {
        mcpsDataReq_t       dataReq;
        mcpsPurgeReq_t      purgeReq;
    } msgData;
} nwkToMcpsMessage_t;
```

4.3. MAC layer configuration

4.3.1. Task configuration

The Task Management System is not in the scope of this document because it is part of the Connectivity Framework.

4.3.2. Memory configuration

The Memory Management System is not in the scope of this document because it is part of the Connectivity Framework.

4.3.3. Message system API

The Message System is not in the scope of this document because it is part of the Connectivity Framework.

5. Interfacing to the 802.15.4 MAC software

This section describes how to interface an application to the MAC, and how to use the MAC interface functions. The examples shown in this section explain the API functions, and they are often simplified for this purpose. See *Freescale 802.15.4 Media Access Controller (MAC) Demo Applications User's Guide* (document 802154MPDAUG) for a detailed walk-through on how to create a more advanced application. A brief overview of the MAC interfaces is provided in the Interface overview sections before describing each interface in more detail. Throughout this section, the term "application" refers to the next higher layer and all layers above the MAC layer. This can be a ZigBee Network, an application, or another network layer written directly on top of the MAC layer.

5.1. Interface overview

The interface between the application layer and the MAC layer is based on service primitives, passed as messages from one layer to another. These service primitives are implemented as a number of C structures with fields for command opcodes/identifiers and command parameters. This section does not describe the primitives in detail, but focuses on the functions used for passing, receiving, and processing the message primitives. For a description of all available message primitives, see Section 4, "MAC/Network Layer Interface Description."

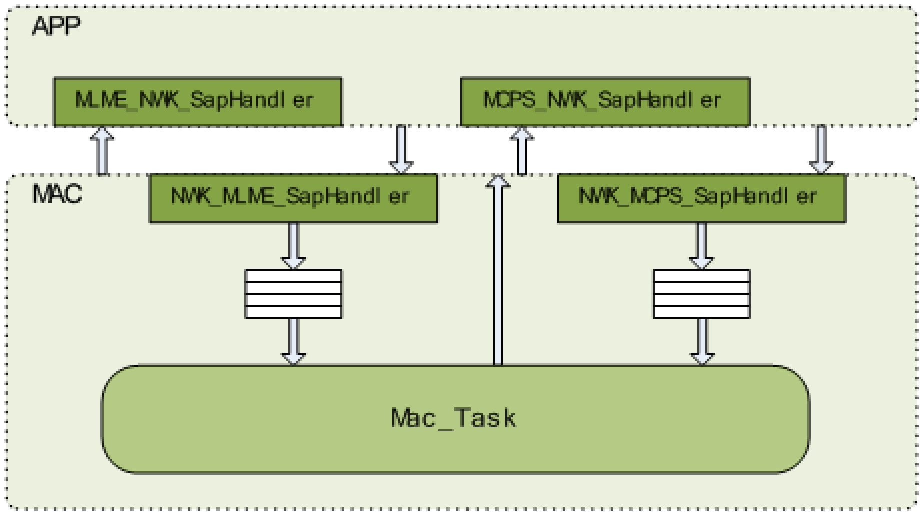

Figure 5. MAC interfaces

Messages are sent to a Service Access Point (SAP) function, which is responsible for handling messages. Five SAPs exists, two for the communication stream to the MAC and three for the communication from the MAC to the application layer.

The following list shows the Service Access Points provided by the MAC layer:

1. `NWK_MLME_SapHandler`—used for command-related primitives sent from the application (or network) layer to the MAC layer. This SAP receives all MLME request and response primitives. See Section 5.5, “MAC main task” for a detailed description.
2. `NWK_MCPS_SapHandler`—used for data-related primitives sent from the application layer to the MAC layer. This SAP receives all MCPS request and response primitives. See Section 5.5, “MAC main task” for a detailed description.

The following list shows the Service Access Points (SAPs) that must be implemented into the application layer by the MAC user:

1. `MLME_NWK_SapHandler_t`—type function—used for command-related primitives sent from the MAC layer to the application/network layer. The access point receives all MLME confirm and indication primitives. See Section 5.6, “MLME and MCPS interface” for a detailed description.
2. `MCPS_NWK_SapHandler_t`—type function—used for data-related primitives sent from the application layer to the MAC layer. This SAP receives all MCPS confirm and indication primitives. See Section 5.6, “MLME and MCPS interface” for a detailed description.

NOTE

If the Multiple Pan feature is used, the discrimination between PANs is done by indicating the specific `xxxInstanceId`.

The application and MAC SAPs usually store the received messages in message queues. A message queue decouples the execution context, which ensures that the call stack does not build up between modules when communicating. The decoupling also ensures that timing-critical modules can queue a message to less timing-critical modules and move on, which ensures that the receiving module processes the message immediately.

5.2. Include files

This table shows which MAC files to include into the application C-files to gain access to the entire MAC API.

Table 8. Required MAC include files in application C-files

Include file name	Description
EmbeddedTypes.h	Provides Freescale specific type defines for creating fixed-size variables. For example, uint8_t defines the type of an 8-bit unsigned integer. It also defines the TRUE and FALSE constants.
MacInterface.h	Defines structures and constants for all MLME and MCPS primitives, as well as all the MAC SAP handlers and the MAC generic interface, used for initialization and general configuration of the MAC layer.

Table 8. Required MAC include files in application C-files

Include file name	Description
PhyInterface.h	Defines structures and constants for all PLME and PD primitives, as well as all the PHY SAP handlers and the PHY generic interface, used for initialization and general configuration of the PHY layer.
FunctionLib.h	Generic function library.
GenericList.h	Header file for the linked lists support.
MacGlobals.h	Header file that provides generic MAC/PHY library configuration available at runtime (for example, task priority, table sizes, number of supported MAC instances in multi-pan configuration, external symbols, and so on).
FsciMacCommands.h	Header file for the Test Client commands and primitives.
MpmInterface.h	Header file for the Multiple PAN Manager.

The below table shows which Connectivity Framework files to include into the application C-files to gain access to the entire Connectivity Framework API.

Table 9. Required connectivity framework include files in application C-files

Include File Name	Description
FsciInterface.h	Header file for the Freescale Serial Communication Interface (FSCI) diagnostic tool.
MemManager.h	Header file for the Memory Management Services implemented by the Connectivity Framework.
Messaging.h	Header file containing the type definitions of inter-task Messaging Interface.
NVM_Interface.h	Header file that contains the Non-Volatile Storage Module interface declarations.
Panic.h	Header file for the Panic module.
RNG_Interface.h	Header file for the Random Number Generation API.
SerialManager.h	Header file needed for the usage of Serial Interfaces (UART, USB, SPI, IIC).
TimersManager.h	Header file for the Timing Services with increased resolution.

5.3. Source files

This table shows which MAC source files to include into the application project.

Table 10. Required MAC source files in application project

Source File Name	Description
MacGlobals.c	Provides MAC-specific configurable memory allocation pools.
FsciMacCommands.c	Provides the MAC-specific command adaptation functions needed to perform testing of the protocol layer interfaces and to communicate with the Host application.

5.4. MAC API

The MAC API provides a simple and consistent way to interface to the Freescale 802.15.4 MAC software. The number of API functions that the Freescale MAC software provides to the application is limited to keep the interfaces as simple and consistent as possible. The API functions available are used for initializing the MAC, running the MAC, allocating, deallocating, sending messages, and queuing and dequeuing of messages. This table shows the available API functions for initializing and running the MAC.

Table 11. Available API functions

Function	Description
MAC_Init	This function initializes the internal variables of the MAC/PHY modules, resets the state machines, and so on. When the function is called, the MAC layer services are available and the MAC and PHY layers are in a known and ready state for further access.
Mac_Task	Because the MAC is designed to be independent of an operating system, part of the MAC must be executed regularly by the MAC task. This task processes the data/command messages that are pending in any of the MAC input queues. See Section 5.5, “MAC main task” for an in-depth description of the Mac_Task function.
BindToMAC	This function creates logical binding with the next available (un-binded) MAC instance.
UnBindFromMAC	This function removes the logical bind of the specified MAC instance.
Mac_GetState	This function checks the states of all MAC instances and returns a general MAC status.
Mac_RegisterSapHandlers	This function registers the MCPS and MLME SAPs, offering support for the MCPS and MLME-to-NWK message interactions.
Mac_GetMaxMsduLength	This function returns the maximum MSDU that can be accommodated considering the current PHY configuration.

To allocate and deallocate messages to and from the MAC, and to send messages to the MAC, use the message handling functions provided by the Connectivity Framework, and described in the *Connectivity Framework Reference Manual*.

Table 12. Exposed message handling functions

Function	Description
MEM_BufferAlloc	Allocates a message
MEM_BufferFree	Frees a message

The Connectivity Framework also implements functions for queuing and dequeuing of messages from the MAC to the application. These functions are shown in this table:

Table 13. Queuing and dequeuing functions

Function	Description
MSG_InitQueue(*pAnchor)	Initializes a queue. Call this function before queuing or dequeuing from the queue.
status = MSG_Pending(*pAnchor)	Checks if a message is pending in a queue given a queue anchor. Returns TRUE if any pending messages, and FALSE otherwise.
MSG_Queue(*pAnchor, *pMsg)	Queues a message given a queue anchor and a pointer to the message.
*pMsg = MSG_DeQueue(*pAnchor)	Gets a message from a queue. Returns NULL if there are no messages in the queue.

Some data types are available also on the MLME and MCPS interfaces. The common element of data structures is that a member variable must be set to indicate which message is being sent. The rest of the data structure is a union that must be accessed and set up accordingly.

Table 14. Data structures passed from the application

Data type	Description
mlmeMessage_t	The data structure of messages passed from the application to the MLME SAP handler.
nwkToMcpsMessage_t	The data structure of messages passed from the application to the MCPS SAP handler.

Similar data structures are used when the MAC sends messages to the application. Again, a member variable contains the message type, and the rest of the data structure is a union that must be decoded accordingly.

Table 15. Data structures passed from the MAC

Data type	Description
nwkMessage_t	The data structure of messages passed from the MLME to the application's MLME SAP handler.
mcpsToNwkMessage_t	The data structure of messages passed from the MCPS to the application's MCPS SAP handler.

The helper structure for managing message queues is also defined.

Table 16. Helper structures for managing message queues

Data type	Description
anchor_t	Container for any type of MAC message. Before queuing or dequeuing messages into the structure, initialize the anchor using `MSG_InitQueue()`. Queue and dequeue the messages using `MSG_Queue()` and `MSG_DeQueue()`.

5.5. MAC main task

Because the MAC is designed to be independent of an operating system, part of the MAC is executed by the MAC task with the root function `Mac_Task()`. The MAC task is responsible for these functions:

- Restructuring data and command frames from the application to the 802.15.4 MAC packet format and vice versa. This includes processing all primitives sent to the MLME and MCPS SAPs.
- Matching data requests received from remote devices against the packets queued for indirect transfer.
- Processing the GTS fields, pending address fields, and the beacon payload of received beacon frames.
- Automatically generating data request packets to extract pending data from remote devices (only in beacon mode and when the PIB attribute macAutoRequest is set to TRUE).
- Applying encryption/decryption to the MAC frames if security is enabled.

Even though these activities are not time-critical in nature, the application program must execute this function in a timely manner. Here are the specific requirements for this execution:

- Execute the MAC main task at least once for every Rx packet the application wants to receive. If the MAC is in a state where the receiver is enabled either continuously or with a regular interval, packets can be expected “anytime,” and the worst-case packet-receive latency is increased with the interval at which the application executes the Mac_Task function. The MAC does not crash if all receive buffers fill up due to slow Mac_Task polling, but the receiver switches off even though it must be enabled.

- Execute the function at least once for every MCPS or MLME primitive the application sends to one of the MAC access points.
- Execute the function at least once between two beacon receptions to ensure basic beacon operation. If this requirement is not met, the beacon packets are not processed in a timely manner, and this can cause unexpected behavior. For optimal beacon operation, it is recommended to execute the MAC main task at least twice during every superframe.

5.6. MLME and MCPS interfaces

The MLME and MCPS interfaces are similar to each other in the way that they are used, and both of them are specified in the 802.15.4 Standard. The MLME interface manages all commands, responses, indications, and confirmations used for managing PAN and 802.15.4-compliant unit. The MCPS interface carries data-related messages (data requests, data indications, data confirmations) and the number of available messages is much smaller than on the MLME interface.

5.6.1. Resetting

Before accessing the Freescale 802.15.4 MAC layer after power-on, initialize it by calling the `MAC_Init` function. At any point after this, it is always safe to reset the MAC (and also the PHY) layer using the service MLME-RESET.Request, as shown in this example:

Example 1. MAC reset example

```
resultType_t App_MacReset_Example(void)
{
    instanceId_t instanceId = 0;
    mlmeMessage_t mlmeReset;

    /* Create and execute the Reset request */
    mlmeReset.msgType = gMlmeResetReq_c;
    mlmeReset.msgData.resetReq.setDefaultPib = TRUE;
    return NWK_MLME_SapHandler(&mlmeReset, instanceId);
}
```

The `setDefaultPib` parameter tells the MAC layer whether its PIB attributes must be set to their default value, or stay unchanged after the reset. This is specified in the 802.15.4 Standard. Notice that the reset is processed immediately (in the `NWK_MLME_SapHandler()`), and the Freescale MAC does not generate the confirmation message MLME-RESET.Confirm. You don't need to check the return value of `NWK_MLME_SapHandler()` as it always returns gSuccess_c on MLME-RESET.Request. Because the call is processed immediately, you do not need to allocate the message structure through `MEM_BufferAlloc()`, but allocate the message structure on the stack. The calling entity has to deallocate the message for MLME-RESET.Request.

NOTE

The MLME-RESET.Request only acts on the PAN settings indicated by the `instanceId` parameter.

5.6.2. Accessing PIB attributes

The MAC PIB holds all the MAC attributes/variables that are accessible for the application. According to the 802.15.4 Standard, the primitives MLME-SET.Request and MLME-GET.Request provide access to the PIB. Similar to the Freescale implementation of MLME-RESET, these primitives are processed immediately, and the corresponding confirmation messages MLME-SET.Confirm and MLME-GET.Confirm are not generated. Instead, the return code from `NWK_MLME_SapHandler()` is used to check the status. The MLME-SET.Request message contains a pointer to the data to be written to the MAC PIB, which must be supplied by the application. This is an example of using MLME-SET.Request:

Example 2. Setting MAC PIBs

```
resultType_t App_SetMacPib_Example(pibId_t attribute, void *pValue)
{
    instanceId_t instanceId = 0;
    mlmeMessage_t mlmeSet;

    /* Create and execute the Set request */
    mlmeSet.msgType = gMlmeSetReq_c;
    mlmeSet.msgData.setReq.pibAttribute = attribute;
    mlmeSet.msgData.setReq.pibAttributeIndex = 0;
    mlmeSet.msgData.setReq.pibAttributeValue = pValue;

    return NWK_MLME_SapHandler(&mlmeSet, instanceId);
}
```

This usage is similar to MLME-RESET.Request, because the call is processed immediately.
The message structure does not need to be allocated through `MEM_BufferAlloc()`, but it can be allocated on the stack. The calling entity must deallocate the message (and potentially the PIB attribute value data buffer) for MLME-SET.Request.

The return value of `NWK_MLME_SapHandler()` is gSuccess_c (when the set request was processed correctly) or gInvalidParameter_c (if the parameter verification failed). In the latter case, the PIB attribute is not set to the new value. The use of MLME-GET.Request is similar, and the only difference is that the `pibAttributeValue` parameter in the message is a return value, and that the value of `msgType` is different.

5.6.3. MLME primitives

The MLME-SET.Request, MLME-GET.Request, and MLME-RESET.Request are the only messages that are completed synchronously. All other messages from the application to the MLME interface are completed asynchronously; meaning that a confirmation message is generated in the MLME and sent to the MLME SAP handler of the application (`MLME_NWK_SapHandler()`).

For example, to request an energy detection scan, the application allocates a MLME message using `MEM_BufferAlloc(sizeof(mlmeMessage_t)`, fills out the parameters of the message (MLME-SCAN.Request), and sends the message to the MLME SAP handler, as shown in the following code example. For a detailed explanation of energy detection scan, see Section 6.2.2, "Energy detection scan".

Example 3. MAC start example

```
uint8_t App_StartEdScan_Example(void)
{
    mlmeMessage_t *pMsg;

    /* Allocate a message for the MLME. */
    pMsg = MEM_BufferAlloc(sizeof(mlmeMessage_t));
    if( pMsg != NULL )
    {
        /* Allocation succeeded. Fill out the message */
        pMsg->msgType = gMlmeScanReq_c;
        pScanReq->msgData.scanReq.scanType = gScanModeED_c;
        pScanReq->msgData.scanReq.scanChannels = 0x07FFF800;
        pScanReq->msgData.scanReq.scanDuration = 5;
        pScanReq->msgData.scanReq.securityLevel = gMacSecurityNone_c;
        /* Send the Scan request to the MLME. */
        if(NWK_MLME_SapHandler( pMsg, mMacInstance ) == gSuccess_c)
        {
            return errorNoError;
        }
        else
        {
            return errorInvalidParameter;
        }
    }
    else
    {
        /* Allocation of a message buffer failed. */
        return errorAllocFailed;
  }
```

When requesting a service that is completed asynchronously, it is the responsibility of the MLME to deallocate the message. For the application to be able to receive the MLME-SCAN.Confirm message from the MLME, the application implements the MLME SAP handler. This SAP handler only queues the message (which is of `nwkMessage_t` type), and returns as soon as possible. An event is passed to the application task to notify it that a new message from the MLME has arrived. To tell the MLME that the message is received successfully, the SAP handler returns a status code of gSuccess_c (any other return code indicates failure), as shown in this code example:

Example 4. MLME SAP handler implementation example

```
resultType_t MLME_NWK_SapHandler (nwkMessage_t* pMsg, instanceId_t instanceId)
{
    /* Put the incoming MLME message in the applications input queue. */
    MSG_Queue(&mMlmeNwkInputQueue, pMsg);
    OSA_EventSet(&mAppEvent, gAppEvtMessageFromMLME_c);
    return gSuccess_c;
}
```

NOTE

If using the Multiple Pan mode, it is recommended to use separate queues for MLME_NWK SAPs (using the `instanceId` parameter) to differentiate the messages.

As shown in the following code example, the application task checks whether it received the event sent by the `MLME_NWK_SapHandler`. Dequeuing of messages that were received from either the MCPS or MLME interface is done using `MSG_DeQueue()`.

Example 5. Handling messages from MAC

```
void AppThread(void const *argument)
{
    event_flags_t ev;
    void *pMsgIn;
    uint8_t *pEdList;

    while(1)
    {
        OSA_EventWait(&mAppEvent, 0x00FFFFFF, FALSE, OSA_WAIT_FOREVER, &ev);

        /* Try to get a message from the MLME. */
        if( ev & evtMessageFromMLME )
        {
            pMsg = MSG_DeQueue(&mMlmeNwkInputQueue);
            /* Check for a scan confirm message. */
            if( pMsg->msgType == gMlmeScanCnf_c )
            {
                /* Find the pointer to the list of detected energies */
                pEdList = pMsg->msgData.scanCnf.resList.pEnergyDetectList;
                /* Do application specific operation on the detected energies */
                /* […] */
                /* The list of detected energies MUST be freed. */
                /* Note: This is a special exception for scan. */
                MEM_BufferFree(pEdList);
            }

            /* Ensure to always free the message */
            MEM_BufferFree(pMsg);
            pMsg = NULL;
        }
    }
}
```

NOTE

The application frees the MLME message after processing it, and some messages contain pointers to data structures that must be freed before freeing the message itself (as in the case shown in the previous example code). Neglecting to free such data structures (and the messages themselves) causes memory leaks.

The MLME interface only allows for one request to be pending at a time. After you send a scan request message to the MLME, you must wait for a scan confirmation message on the MLME SAP handler before you can send another MLME request to the MLME. Not following this rule may result in unwanted and/or unpredictable behavior. When operating in Multiple Pan mode, the MLME enables a corresponding number of requests pending at the same time if the requests originated from different PANs. If you call a MLME primitive with an invalid argument, the MAC does not pass it down to the lower layers, so the application does not have to wait for a confirmation. In this case `NWK_MLME_SapHandler()` returns the gInvalidParameter_c value. This approach simplifies the application state machine.

5.6.4. MCPS primitives

Similarly to the MLME interface, the MCPS interface processes data-related messages. The MCSP interface is used just like the MLME interface. The main difference is that the message types are sent back and forth on the interface. On the MCPS, use `MEM_BufferAlloc(sizeof(nwkToMcpsMessage_t) + gPhyMaxDataLength_c)` to allocate a MCPS message, as shown in this code example:

Example 6. Example of MCPS-DATA.Request

```
void SendMyName(void)
{
    nwkToMcpsMessage_t *pMsg;
    instanceId_t instanceId = 0;
    uint8_t FSL[] = "Freescale";

    pMsg = MEM_BufferAlloc(sizeof(nwkToMcpsMessage_t) + gPhyMaxDataLength_c)
    if( pMsg != NULL )
    {
        pMsg->msgType = gMcpsDataReq_c;
        /* initialize pointer to Msdu */
        pMsg->msgData.dataReq.pMsdu = (uint8_t*)(&(pMsg->msgData.dataReq.pMsdu)) +
sizeof(uint8_t*);
        FLib_MemCpy(pMsg->msgData.dataReq.pMsdu, FSL, sizeof(FSL));
        pMsg->msgData.dataReq.dstAddr = mDstDeviceShortAddress_c;
        pMsg->msgData.dataReq.srcAddr = mSrcDeviceShortAddress_c;
        pMsg->msgData.dataReq.dstPanId = mDstDevicePanId_c;
        pMsg->msgData.dataReq.srcPanId = mSrcDevicePanId_c;
        pMsg->msgData.dataReq.dstAddrMode = gAddrModeShortAddress_c;
        pMsg->msgData.dataReq.srcAddrMode = gAddrModeShortAddress_c;
        pMsg->msgData.dataReq.msduLength = sizeof(FSL);
        /* Add Indirect Transmission option in order to send to polling end device */
        pMsg->msgData.dataReq.txOptions = gMacTxOptionsAck_c | gMacTxOptionIndirect_c;
        pMsg->msgData.dataReq.msduHandle = mMsduHandle++;
        /* Send the Data Request to the MCPS */
       (void)NWK_MCPS_SapHandler(pMsg, instanceId);
    }
}
```

The application is allowed to allocate multiple data messages using `MEM_BufferAlloc([…])` until it returns NULL. Because the MCPS interface can manage multiple outstanding data requests, it is possible to use multiple buffering for maximum throughput. You can send a data request to the MCPS, and then, before receiving data confirmation on that request, send another data request which keeps a constant data flow between the application and the MCPS interface.

Even though it is not supported by the 802.15.4 Standard, you can use double buffering also for polling. See Section 6.11, "Data primitives" for optimizing data throughput using double buffering.

When the application receives messages from the MCPS, the messages are received as `mcpsToNwkMessage_t` type in the `MCPS_NWK_SapHandler()` function (as shown in the following code example).

Example 7. MCPS SAP handler implementation example

```
resultType_t MCPS_NWK_SapHandler(mcpsToNwkMessage_t* pMsg, instanceId_t instanceId)
{
    /* Put the incoming MCPS message in the applications input queue. */
    MSG_Queue(&mMcpsNwkInputQueue, pMsg);
    OSA_EventSet(&mAppEvent, gAppEvtMessageFromMCPS_c);
    return gSuccess_c;
}
```

The MCPS SAP handler in the application is similar to that of MLME. However, use separate queues for the MCPS and MLME messages, because the messages cannot be differentiated when the SAP handler has finished and the messages have been queued.

NOTE

If using Multiple Pan, it is recommended that you use separate queues for MLME_NWK SAPs (using the `instanceId` parameter) to differentiate the messages.

The MCPS message processing is performed similarly to the processing of MLME messages (typically in the application task) and the application is responsible for deallocating the MCPS messages.

6. Feature Descriptions

The section describes the Freescale 802.15.4 MAC/PHY software features, focusing on the implementation-specific details of the 802.15.4 2006 (*0*) and 2011 (*0*) Standards. See the appropriate 802.15.4 Standard for more details on these features.

NOTE

The differences between the 802.15.4 Standard (2006) and the 802.15.4 Standard (2011) relevant to the MAC software are noted where appropriate.

6.1. Configuration

The MAC contains a programmable PAN Information Base (PIB). It consists of variables controlling the operation of the MAC. Some of the variables are updated by the MAC, while others are configured by the upper layer. The MAC PIB attributes, and the three primitives available for configuring the MAC PIB are described in the following sections.

6.1.1. PIB attributes

This table shows all available MAC PIB attributes, including Freescale-specific additions to the 802.15.4 Standard-specific attributes:

Table 17. Available PIB attributes

PIB attribute	Description
Freescale-specific attributes	
0x20	gMPibRole_c—contains the current role of the device: • 0x00 = Device • 0x01 = Coordinator • 0x02 = PAN Coordinator
0x21	gMPibLogicalChannel_c—contains the current logical channel (11 to 26).
0x22	gMPibTreemodeStartTime_c—used to support Beaconed Tree Mode (Legacy).
0x23	gMPibPanIdConflictDetection_c—disables or enables PAN ID conflict detection (Legacy).
0x24	gMPibBeaconResponseDenied_c—if set on TRUE, the coordinator does not respond to beacon requests (Legacy).
0x25	gMPibNBSuperFrameInterval_c—the length of the superframe for non-beacon mode. This attribute is used instead of gMPibBeaconOrder_c to calculate the persistence time of indirect packets when the coordinator runs in non-beacon mode. See the IEEE 802.15.4 Standard for details on calculating the persistence time (Legacy).
0x26	gMPibBeaconPayloadLengthVendor—vendor-specific Beacon Payload length (Legacy).
0x27	gMPibBeaconResponseLQIThreshold_c—used to filter incoming beacon requests based on their LQI value. 0x00 (default) = filter is disabled. The coordinator responds to all incoming beacon requests. 0x01-0xFF = filter is enabled. The coordinator does not respond to incoming beacon requests that have the LQI smaller than the configured threshold (Legacy).
0x28	gMPibUseExtendedAddressForBeacon_c—if set to TRUE, MAC layer sends beacon frames using extended address.
0x30	gMacPibExtendedAddress_c—used to store the device extended address.
802.15.4-specific attributes (see 0 for descriptions)	
0x40	gMPibAckWaitDuration_c—the maximum number of symbols to wait for an acknowledgment frame to arrive, following a transmitted data frame.

Table 17. Available PIB attributes

PIB attribute	Description
0x41	gMPibAssociationPermit_c—indication of whether a coordinator is currently allowing association.
0x42	gMPibAutoRequest_c—indication of whether a device automatically sends data request command (if its address is listed in the beacon frame).
0x43	gMPibBattLifeExt_c—indication of whether the BLE is enabled, using the reduction of coordinator receiver operation time during the CAP.
0x44	gMPibBattLifeExtPeriods_c—in the BLE mode, the number of backoff periods during which the receiver is enabled after the IFS following a beacon.
0x45	gMPibBeaconPayload_c—the contents of the beacon payload.
0x46	gMPibBeaconPayloadLength_c—the length of the beacon payload (in octets).
0x47	gMPibBeaconOrder_c—specifies how often the coordinator transmits its beacon.
0x48	gMPibBeaconTxTime_c—the time when the device transmitted its last beacon frame (in symbol periods). Take the measurement at the same symbol boundary within every transmitted beacon frame (the location of the transmitted beacon frame is implementation-specific).
0x49	gMPibBsn_c—the sequence number added to the transmitted beacon frame.
0x4A	gMPibCoordExtendedAddress_c—the 64-bit address of the coordinator, through which the device is associated.
0x4B	gMPibCoordShortAddress_c—the 16-bit short address assigned to the coordinator, through which the device is associated.
0x4C	gMPibDsn_c—the sequence number added to the transmitted data or MAC command frame.
0x4D	gMPibGtsPermit_c—indication of whether a PAN coordinator is to accept GTS requests.
0x4E	gMPibMaxCsmaBackoffs_c—the maximum number of backoffs that the CSMA-CA algorithm attempts before declaring channel access failure.
0x4F	gMPibMinBe_c—the minimum value of the Backoff Exponent (BE) in the CSMA-CA algorithm.
0x50	gMPibPanId_c—the 16-bit identifier of the PAN, on which the device operates.
0x51	gMPibPromiscuousMode_c—indication of whether the MAC sublayer is in a promiscuous (receive all) mode.
0x52	gMPibRxOnWhenIdle_c—indication of whether the MAC sublayer is to enable its receiver during idle periods.
0x53	gMPibShortAddress_c—the 16-bit address that the device uses to communicate in the PAN.
0x54	gMPibSuperFrameOrder_c—the length of the active portion of the outgoing superframe, including the beacon frame.
0x55	gMPibTransactionPersistenceTime_c—the maximum time (in unit periods), for which a transaction is stored by a coordinator and indicated in its beacon.
0x56	gMPibAssociatedPANCoord_c—indication of whether the device is associated to the PAN through the PAN coordinator.

Table 17. Available PIB attributes

PIB attribute	Description
0x57	gMPibMaxBe_c—the maximum value of the Backoff Exponent (BE) in the CSMA-CA algorithm.
0x59	gMPibMacFrameRetries_c—the maximum number of retries allowed after a transmission failure.
0x5A	gMPibResponseWaitTime_c—the maximum time (in multiples of aBaseSuperframeDuration) a device must wait for a response command frame to be available following a request command frame.
0x5B	gMPibSyncSymbolOffset_c—the offset (measured in symbols) between the symbol boundary at which the MLME captures the timestamp of each transmitted or received frame, and the onset of the first symbol past the SFD, namely, the first symbol of the Length field.
0x5C	gMPibTimeStampSupported_c—indication of whether the MAC sublayer supports the optional timestamping feature for incoming and outgoing data frames.
0x5D	gMPibSecurityEnable_c—indication of whether the MAC sublayer has security enabled.
0x5E	gMPibMinLIFSPeriod_c—minimum accepted value for Long Interframe Spacing (LIFS) period.
0x5F	gMPibMinSIFSPeriod_c—minimum accepted value for Short Interframe Spacing (SIFS) period.
0x60	gMPibTxControlActiveDuration_c—802.15.4-2011-specific attribute. This value controls the time, for which a device may transmit, when using Listen Before Talk (LBT).
0x61	gMPibTxControlPauseDuration_c—802.15.4-2011-specific attribute. This value controls the duration of the pause period (the time during which the MAC pauses to enable other devices to access the channel) when using Listen Before Talk (LBT).
0x62	gMPibTxTotalDuration_c—802.15.4-2011-specific attribute. This value controls the transmission duty cycle of the operation when using Listen Before Talk (LBT).
CSL (Coordinated Sampled Listening)-specific attributes	
0x63	gMPibCslPeriod_c—CSL sampled listening period (in unit of 10 symbols).
0x64	gMPibCslMaxPeriod_c—maximum CSL sampled listening period in the entire PAN (in unit of 10 symbols).
0x65	gMPibCslFramePendingWait_c—number of symbols to keep the receiver on after receiving a payload frame with Frame Control field frame pending bit set to one.
0x66	gMPibEnhAckWaitDuration_c—the maximum time (in µs) to wait for the PHY header of an Enhanced Acknowledgment frame to arrive following a transmitted data frame.
RIT (Receiver Initiated Transmission)-specific attributes	
0x67	gMPibRitPeriod_c—the interval (in unit periods) for periodic transmission of RIT data request command in RIT mode.
0x68	gMPibRitDataWaitDuration_c—the maximum time (in unit period) to wait for Data frame after transmission of RIT data request command frame in RIT mode.
0x69	gMPibRitTxWaitDuration_c—the maximum time (in unit periods) that a transaction is stored by a device in RIT

Table 17. Available PIB attributes

PIB attribute	Description
	mode.
0x6A	gMPibRitIe_c—the structure of the Low Energy RIT IE.
Sub-1 GHz-specific attributes	
0x6B	gMacPibPhyMode_c—available PHY modes: • 0x00 = gPhyMode1_c • 0x01 = gPhyMode2_c • 0x02 = gPhyMode3_c • 0x03 = gPhyMode4_c • 0x04 = gPhyMode1ARIB_c • 0x05 = gPhyMode2ARIB_c • 0x06 = gPhyMode3ARIB_c
0x6C	gMacPibPhyCCADuration_c—the CCA Duration (in symbols).
0x6D	gMacPibPhyFSKScramblePSDU_c—flag to enable/disable PSDU data whitening. For MR-FSK PHY only.
0x6E	gMacPibPhyFrequencyBand_c—SUN PHY frequency band definitions.
TSCH (Time Slotted Channel Hopping)-specific attributes	
0x31	gMPibDisconnectTime_c—time (in timeslots) to send disassociation frames before disconnecting (Reserved).
0x32	gMPibJoinPriority_c—join priority as set in the TSCH Synchronization IE in Enhanced beacon.
0x33	gMPibASN_c—Absolute Slot Number, the number of timeslots elapsed since the network has been started, also used as CCM* when TSCH and security are enabled.
0x34	gMPibNoHLBuffers_c—if set to TRUE, the higher layer indicates that the received data frame cannot be buffered. The MAC layer then indicates this to peer by filling the ACK/NACK accordingly.
0x35	gMPibEBSN_c—the Enhanced beacon frame sequence number, separated from the beacon sequence number.
0x36	gMPibTschEnabled_c—indicates whether the TSCH module is enabled.
0x37	gMPibEBIEList_c—the Enhanced beacon frame list of Information Elements, as included in the frame when the MLME-BEACON.Request is issued. Set this PIB before sending the primitive to the MAC layer. The available IE IDs are: • 0x09 = gMacPayloadIeIdChannelHoppingSequence_c • 0x1A = gMacPayloadIeIdTschSynchronization_c • 0x1B = gMacPayloadIeIdTschSlotframeAndLink_c • 0x1C = gMacPayloadIeIdTschTimeslot_c
0x38	gMPibTimeslotTemplate_c—the timeslot template specifying timings to be used inside the timeslot for CCA, Tx/Rx of frame, Tx/Rx of Ack, and so on.
0x39	gMPibHoppingSequenceList_c—list of channels to perform hopping on, having the length set at gMPibHoppingSequenceLength_c.

Table 17. Available PIB attributes

PIB attribute	Description
0x3A	gMPibHoppingSequenceLength_c—length of the Channel Hopping sequence.
0x3B	gMPibTschRole_c—the TSCH Role of a node, set for either the device that coordinates the TSCH PAN or joins a TSCH PAN • 0x00 = gMacTschRolePANCoordinator_c • 0x01 = gMacTschRoleDevice_c
Security-specific attributes	
0x71	gMPibKeyTable_c—a table of KeyDescriptor entries, each containing keys and related information required for secured communications.
0x72	gMPibKeyTableEntries_c—the number of entries in macKeyTable. This PIB is read-only. Freescale-specific attribute for Std. 802.15.4-2011.
0x73	gMPibDeviceTable_c—a table of DeviceDescriptor entries, each indicating a remote device with which this device securely communicates.
0x74	gMPibDeviceTableEntries_c—the number of entries in macDeviceTable. This PIB is read-only. Freescale-specific attribute for Std. 802.15.4-2011.
0x75	gMPibSecurityLevelTable_c—a table of SecurityLevelDescriptor entries, each with information about the minimum security level expected depending on incoming frame type and subtype.
0x76	gMPibSecurityLevelTableEntries_c—the number of entries in macSecurityLevelTable. Freescale-specific attribute for Std. 802.15.4-2011.
0x77	gMPibFrameCounter_c—the outgoing frame counter for this device.
0x78	gMPibAutoRequestSecurityLevel_c—the security level used for automatic data requests.
0x79	gMPibAutoRequestKeyIdMode_c—the key identifier mode used for automatic data requests. This attribute is invalid if the macAutoRequestSecurityLevel attribute is set to 0x00.
0x7A	gMPibAutoRequestKeySource_c—the originator of the key used for automatic data requests. This attribute is invalid if the macAutoRequestKeyIdMode element is invalid or set to 0x00.
0x7B	gMPibAutoRequestKeyIndex_c—the index of the key used for automatic data requests. This attribute is invalid if the macAutoRequestKeyIdMode attribute is invalid or set to 0x00.
0x7C	gMPibAutoDefaultKeySource_c—the originator of the default key used for key identifier mode 0x01.
0x7D	gMPibPANCoordExtendedAddress_c—the 64-bit address of the PAN coordinator.
0x7E	gMPibPANCoordShortAddress_c—the 16-bit short address assigned to the PAN coordinator. A value of 0xfffe indicates that the PAN coordinator is only using its 64-bit extended address. A value of 0xffff indicates that this value is unknown.

Table 17. Available PIB attributes

PIB attribute	Description
Freescale security-specific attributes	
0x7F	gMPibKeyIdLookupDescriptor_c—a list of KeyIdLookupDescriptor entries used to identify this KeyDescriptor.
0x80	gMPibKeyIdLookupListEntries_c—the number of entries in the KeyIdLookupList. Std. 802.15.4-2011-specific attribute.
0x81	gMPibKeyDeviceList_c—a list of KeyDeviceDescriptor entries indicating which devices are currently using this key, including their blacklist status.
0x82	gMPibKeyDeviceListEntries_c—the number of entries in KeyDeviceList.
0x9C	gMPibDeviceDescriptorHandleList_c—Std. 802.15.4-2011-specific attribute. A list of implementation-specific handles to DeviceDescriptor entries in macDeviceTable for each of the devices that currently use this key.
0x9D	gMPibDeviceDescriptorHandleListEntries_c—Std. 802.15.4-2011-specific attribute. The number of entries in macDeviceTable.
0x83	gMPibKeyUsageList_c—a list of KeyUsageDescriptor entries indicating which frame types this key can be used with.
0x84	gMPibKeyUsageListEntries_c—the number of entries in KeyUsageList. Std. 802.15.4-2011-specific attribute.
0x85	gMPibKey_c—the actual value of the key.
0x86	gMPibKeyUsageFrameType_c—the type of the frame to secure.
0x87	gMPibKeyUsageCmdFrameId_c—the ID of the command frame to secure.
0x88	gMPibKeyDeviceDescriptorHandle_c—handle to the DeviceDescriptor corresponding to the device.
0x89	gMPibUniqueDevice_c—indicator as to whether the device indicated by DeviceDescriptorHandle is uniquely associated with the KeyDescriptor; it is a link key as opposed to a group key.
0x8A	gMPibBlackListed_c—indicator as to whether the device indicated by DeviceDescriptorHandle previously communicated with this key before the exhaustion of the frame counter. If TRUE, it indicates that the device must not use this key further, since it exhausted its use of the frame counter used with this key.
0x9E	gMPibDeviceDescriptorHandle_c—Std. 802.15.4-2011-specific attribute. Element of DeviceDescriptorHandleList.
0x8B	gMPibSecLevFrameType_c—the type of frame to secure.
0x8C	gMPibSecLevCommandFrameIdentifier_c—the ID of the command frame to secure.
0x8D	gMPibSecLevSecurityMinimum_c—the minimum security level required/expected for incoming MAC frames with the indicated frame type and command frame type (if present).
0x8E	gMPibSecLevDeviceOverrideSecurityMinimum_c—indicates whether the originating devices for which the Exempt flag is set can override the minimum security level indicated by the SecurityMinimum element.

Table 17. Available PIB attributes

PIB attribute	Description
0x9F	gMPibSecLevAllowedSecurityLevels_c—Std. 802.15.4-2011-specific attribute. A set of allowed security levels for incoming MAC frames with the indicated frame type, and command frame identifier (if present).
0x8F	gMPibDeviceDescriptorPanId_c—the 16-bit PAN identifier of the device in this DeviceDescriptor.
0x90	gMPibDeviceDescriptorShortAddress_c—the 16-bit short address of the device in this DeviceDescriptor. A value of 0xfffe indicates that this device is only using its extended address. A value of 0xffff indicates that this value is unknown.
0x91	gMPibDeviceDescriptorExtAddress_c—the 64-bit IEEE extended address of the device in this DeviceDescriptor. This element is also used in unsecuring operations on incoming frames.
0x92	gMPibDeviceDescriptorFrameCounter_c—the incoming frame counter of the device in this DeviceDescriptor. This value is used to ensure sequential freshness of frames.
0x93	gMPibDeviceDescriptorExempt_c—indicator as to whether the device can override the minimum security level settings.
0x94	gMPibKeyIdLookupData_c—data used to identify the key.
0x95	gMPibKeyIdLookupDataSize_c—a value of 0x00 indicates a set of five octets; a value of 0x01 indicates a set of nine octets.
0xA0	gMPibKeyIdLookupKeyIdMode_c—Std. 802.15.4-2011-specific attribute. The mode used to for this descriptor.
0xA1	gMPibKeyIdLookupKeySource_c—Std. 802.15.4-2011-specific attribute. Information to identify the key.
0xA2	gMPibKeyIdLookupKeyIndex_c—Std. 802.15.4-2011-specific attribute. Information used to identify the key.
0xA3	gMPibKeyIdLookupDeviceAddressMode_c—Std. 802.15.4-2011-specific attribute. The addressing mode for this descriptor.
0xA4	gMPibKeyIdLookupDevicePANId_c—Std. 802.15.4-2011-specific attribute. The PAN identifier for this descriptor.
0xA5	gMPibKeyIdLookupDeviceAddress_c—Std. 802.15.4-2011-specific attribute. The device address for this descriptor.
0x96	gMPibiKeyTableCrtEntry_c—the current entry in the macKeyTable.
0x97	gMPibiDeviceTableCrtEntry_c—the current entry in the macDeviceTable.
0x98	gMPibiSecurityLevelTableCrtEntry_c—the current entry in the macSecurityLevelTable.
0x99	gMPibiKeyIdLookupListCrtEntry_c—the current entry in the KeyIdLooupList.
0x9A	gMPibiKeyUsageListCrtEntry_c—the current entry in the KeyUsageList.
0xA6	gMPibiDeviceDescriptorHandleListCrtEntry_c—Std. 802.15.4-2011-specific attribute. The current entry in the DeviceDescriptorHandleList.
0x9B	gMPibiKeyDeviceListEntry_c—the current entry in the KeyDeviceList.

6.1.2. Configuration primitives

This section describes the implementation of the configuration-related primitives.

6.1.2.1. Reset request

The internal state of the MAC (including the messages allocated from the data buffer system) is always reset by the MLME-RESET.Request. However, the upper layer can choose whether the MAC PIB attributes must be set to default values. This is accomplished through the setDefaultPib parameter of the MLME-RESET.Request. If the parameter is TRUE, the MAC PIB is reset to default values, otherwise the contents are left untouched.

The Reset-Request message is processed immediately, and can be allocated on the stack. If the message is allocated by *MEM_BufferAlloc()*, it is not freed by the MLME, and a confirm message is not generated. Instead, the return code from the *NWK_MLME_SapHandler()* function is used as the status code.

```
// Type: gMlmeResetReq_c,
typedef STRUCT mlmeResetReq_tag
{
    bool_t                  setDefaultPIB;
} mlmeResetReq_t;
```

6.1.2.2. Reset-confirm (N/A)

The Reset-confirm message is not used, because the Reset is carried out immediately. The Confirmation status code is returned by the SAP function that sends the Reset-Request message to the MLME.

```
// Type: gMlmeResetCnf_c,
typedef STRUCT mlmeResetCnf_tag
{
    resultType_t            status;
} mlmeResetCnf_t;
```

6.1.2.3. Set request

The MLME-SET.Request is used for modifying parameters in the MAC PIB. See the *802.15.4 MAC/PHY API Reference Manual* (document 802154MPAPIRM), or Section 6.1.1, "PIB attributes" for a list of available PIB attributes.

The Set request message structure contains a pointer to the data to be written to the MAC PIB. The pointer must be supplied by the NWK or APP. Attributes with a size greater than one byte must be little endian, and given as byte arrays. Because the Set-Request message is processed immediately, it can be allocated on the stack. If the message is allocated by *MEM_BufferAlloc()*, it is not freed by the MLME. A confirm message is not generated. Instead, the return code from the *NWK_MLME_SapHandler()* function is used as the status code.

When using the Set-Request for setting the beacon payload, the beacon payload length attribute must be set first. Otherwise, the MLME has no way to tell how many bytes to copy.

The SetRequest structure is:

```
// Type: gMlmeSetReq_c,
typedef STRUCT mlmeSetReq_tag
{
    pibId_t                 pibAttribute;
    uint8_t                 pibAttributeIndex;
    void*                   pibAttributeValue;
} mlmeSetReq_t;
```

6.1.2.4. Set-confirm

The Set-confirm as message is not used because the Set-Request is carried out synchronously. See Section 6.1.2.3, “Set request” for more information.

```
// Type: gMlmeSetCnf_c,
typedef STRUCT mlmeSetCnf_tag
{
    resultType_t            status;
    pibId_t                 pibAttribute;
    uint8_t                 pibAttributeIndex;
} mlmeSetCnf_t;
```

6.1.2.5. Get-request

The MLME-GET.Request reads parameters in the MAC PIB. See the *802.15.4 MAC/PHY API Reference Manual* (document 802154MPAPIRM) or Section 6.1.1, “PIB attributes” for a list of available PIB attributes. The Get-request message contains a pointer to a buffer where data from the MAC PIB are copied to. The pointer must be supplied by the NWK or APP. Attributes with size greater than one byte are little endian and given as byte arrays. Because the Get-Request message is processed immediately, it can be allocated on the stack. If the message is allocated by *MEM_BufferAlloc()*, it is not freed by the MLME. A confirm message is not generated. Instead, the return code from the *NWK_MLME_SapHandler()* function is used as the status code.

The Get-request structure is:

```
// Type: gMlmeGetReq_c,
typedef STRUCT mlmeGetReq_tag
{
    pibId_t                 pibAttribute;
    uint8_t                 pibAttributeIndex;
    void*                   pibAttributeValue; // Not in spec.
} mlmeGetReq_t;
```

6.1.2.6. Get-confirm

Get-confirm is not used as message because the Get-request is carried out synchronously.
See Section 6.1.2.5, “Get-request” for more information.

```
// Type: gMlmeGetCnf_c,
typedef STRUCT mlmeGetCnf_tag
{
    resultType_t            status;
    pibId_t                 pibAttribute;
    uint8_t                 pibAttributeIndex;
    void*                   pibAttributeValue;
} mlmeGetCnf_t;
```

6.1.3. Configuration examples

The following code snippets show examples of sending configuration messages to the MLME.

Example 8. Sending a set-request with macPanId=0x1234

```
#define mDefaultValueOfPanId_c (0x1234)

static const instanceId_t mMacInstance = 0;
static const uint16_t mPanId = mDefaultValueOfPanId_c;

uint8_t confirmStatus;
mlmeMessage_t * pMsg = MEM_BufferAlloc(mlmeMessage_t);
if( pMsg != NULL )
{
    pMsg->msgType = gMlmeSetReq_c;
    pMsg->msgData.setReq.pibAttribute = gMPibPanId_c;
    pMsg->msgData.setReq.pibAttributeValue = (uint8_t*)&mPanId;

    // Calls NWK_MLME_SapHandler(*pMsg, instanceId)
    confirmStatus = NWK_MLME_SapHandler( pMsg, mMacInstance );

    MEM_BufferFree(Msg);
}
```

Example 9. Using the stack instead of the MEM_BufferAlloc() for getting the message buffer

```
mlmeMessage_t msg;
uint8_t autoRequestFlag = TRUE;

// We want to set the PAN ID attribute of the MAC PIB.
msg.msgType = gMlmeSetReq_c;
msg.msgData.setReq.pibAttribute = gMPibPanId_c;
msg.msgData.setReq.pibAttributeValue = (uint8_t*)&mPanId;

// Calls uint8_t NWK_MLME_SapHandler(*pMsg, instanceId)
confirmStatus = NWK_MLME_SapHandler(&msg, mMacInstance);

// Set the MAC PIB Auto request flag to TRUE. No need to set
// message identifier again since the msg is not modified by
// the NWK_MLME_SapHandler(*pMsg, instanceId) call.
msg.msgData.setReq.pibAttribute = gMPibAutoRequest_c;
msg.msgData.setReq.pibAttributeValue = &autoRequestFlag;
confirmStatus = NWK_MLME_SapHandler(&msg, mMacInstance);
```

Example 10. Getting the macBeaconTxTime using the get-request

```
uint8_t txTime[3];

msg.msgType = gMlmeGetReq_c;
msg.msgData.getReq.pibAttribute = gMPibBeaconTxTime_c;
msg.msgData.getReq.pibAttributeValue = txTime;

// Calls uint8_t NWK_MLME_SapHandler(*pMsg, instanceId)
confirmStatus = NWK_MLME_SapHandler(&msg, mMacInstance);

// Now txTime contains the value of macBeaconTxTime
// (24 bit integer in little endian format).
```

6.2. Scan feature

The scan feature is used by the device to determine energy usage or the presence of other devices on a communications channel. This feature is implemented similarly to the 802.15.4 Standard. The specific details are included in this section.

6.2.1. Common parts

Requesting any of the scan types (using the MLME-SCAN.Request primitive) interrupts all other system activities at the MLME layer and below in accordance with the 802.15.4 Standard. It is the responsibility of the NWK layer to initiate scanning when this behavior is acceptable.

The NWK layer is responsible for correct system behavior, particularly by ensuring that only supported scan types are attempted and that at least one channel is always indicated in the ScanChannels parameter.

6.2.2. Energy detection scan

RFD devices and derivatives do not support Energy Detection (ED) scan. When Energy Detection scan is requested, the device measures the energy level on each requested channel until the scan time has elapsed.

The MLME-SCAN.Confirm primitive always holds energy-detection results from all requested channels (partial responses are never returned).

The level for Energy Detection is reported as required by the 802.15.4 Standard with an integer value in the range 0x00-0xFF. The hardware measured values are scaled and normalized for this range with the minimum value of 0x00 representing signal values under the XCVR sensibility, and the maximum value of 0xFF representing values above the XCVR saturation level.

6.2.3. Active and Passive scan

When Active or Passive scans are requested, the device waits for beacons to arrive until the scan time elapses. If a valid unique beacon is received during this period, the device stores the result. In this case, or if any other package was received from the air, the device re-enters Rx mode (as long as there is time for the shortest possible Rx cycle to complete before the complete scan time has elapsed).

Active and Passive scans are capable of returning up to 10 results in a single MLME-SCAN.Confirm primitive. When 10 unique beacons (see the 802.15.4 Standard) are received, the scan is terminated in accordance with the 802.15.4 Standard, even if all channels are not scanned completely.

The PAN descriptors are grouped by five in two linked blocks. The pPanDescriptorBlocks field of the scan confirmation message points to the first block. Each block contains a pointer to the first PAN descriptor in the block and the number of PAN descriptors in that block. The block structure is:

```
typedef STRUCT panDescriptorBlock_tag
{
    panDescriptor_t              panDescriptorList[gScanResultsPerBlock_c];
    uint8_t                      panDescriptorCount;
    struct panDescriptorBlock_tag *pNext;
} panDescriptorBlock_t;
```

6.2.4. Orphan scan

When Orphan scan is requested, the device waits for a coordinator realignment command to arrive until the scan time elapses. If any other command is received from the air during this time, the device ignores the command and re-enters Rx mode (as long as there is time for the shortest possible Rx cycle to complete before the complete scan time elapses).

If a valid coordinator realignment response is received while performing the Orphan scan, scanning is immediately terminated in accordance with the 802.15.4 Standard (*0, 0*), even if all channels are not scanned completely. In this case, the resulting Status parameter is SUCCESS (otherwise NO_BEACON), and MAC PIB attribute values received in the coordinator realignment frame (macPanId, macCoordShortAddress, macLogicalChannel and macShortAddress) are automatically used to update the MAC PIB.

6.2.5. Scan primitives

This section describes the implementation of the Scan-related primitives.

6.2.6. Scan-request

The Scan-request message parameters are directly mapped to the message parameters listed and described in the 802.15.4 Standards (*0, 0*). Ensure that `scanChannels` always indicates at least one valid channel and that channels outside the valid range [11:26] are not indicated. The value of 0x07FFF800 corresponds to “all valid channels.” The valid range for `scanType` is [0:3]. The valid range for `scanDuration` is [0:14].

The Scan-request structure is:

```
// Type: gMlmeScanReq_c,
typedef STRUCT mlmeScanReq_tag
{
    macScanType_t           scanType;
    channelMask_t           scanChannels;
    uint8_t                 scanDuration;
    channelPageId_t         channelPage;
    macSecurityLevel_t      securityLevel;
    keyIdModeType_t         keyIdMode;
    uint64_t                keySource;
    uint8_t                 keyIndex;
} mlmeScanReq_t;
```

- scanType—type of Scan to be carried out.
- scanChannels—bitmask with channels on which Scan is performed.
- scanDuration—scan duration on each channel.
- securityLevel—the security level to be used.
- keyIdMode—this mode identifies the key to be used. This parameter is ignored if the

securityLevel parameter is set to 0.

- keySource—the originator of the key to be used. This parameter is ignored if the keyIdMode parameter is ignored or set to 0x00.
- keyIndex—the index of the key to be used. This parameter is ignored if the keyIdMode parameter is ignored.

6.2.7. Scan-confirm

The Scan-confirm structure contains a pointer to an array of blocks containing PAN descriptors, or a pointer to an array of energy levels. See the definition in Section 6.2.11, “PAN descriptor”. The array must be freed by calling *MEM_BufferFree()* after Energy Detection, Passive, or Active scan. All other parameters map exactly as shown to the parameters listed and described in the 802.15.4 Standard (*0, 0*).

```
// Type: gMlmeScanCnf_c,
typedef STRUCT mlmeScanCnf_tag
{
    resultType_t            status;
    macScanType_t           scanType;
    channelPageId_t         channelPage;
    uint8_t                 resultListSize;
    channelMask_t           unscannedChannels;
    UNION {
        uint8_t*              pEnergyDetectList;
        panDescriptorBlock_t* pPanDescriptorBlockList;
    } resList;
} mlmeScanCnf_t;
```

6.2.8. Orphan-indication

The Orphan-indication structure is the following:

```
// Type: gMlmeOrphanInd_c,
typedef STRUCT mlmeOrphanInd_tag
{
    uint64_t                orphanAddress;
    macSecurityLevel_t      securityLevel;
    keyIdModeType_t         keyIdMode;
    uint64_t                keySource;
    uint8_t                 keyIndex;
} mlmeOrphanInd_t;
```

The security parameter has the same meaning as the Scan-request security parameters.

6.2.9. Orphan-response

The Orphan-response structure is the following:

```
// Type: gMlmeOrphanRes_c,
typedef STRUCT mlmeOrphanRes_tag
{
    uint64_t                orphanAddress;
    uint16_t                shortAddress;
    bool_t                  associatedMember;
    macSecurityLevel_t      securityLevel;
    keyIdModeType_t         keyIdMode;
    uint64_t                keySource;
    uint8_t                 keyIndex;
} mlmeOrphanRes_t;
```

The security parameter has the same meaning as the Scan-request security parameters.

6.2.10. Beacon notify indication

The MLME-BEACON-NOTIFY.Indication message is not only received during scan, but can be also received when the device is tracking a beaconing coordinator.

The MLME-BEACON-NOTIFY.Indication message is special because it contains pointers. The `pAddrList` pointer points to the address list which is formatted according to the 802.15.4 Standard. The `pPanDescriptor` pointer points to the pan descriptor of the indication message. See the definition in Section 6.2.11, "PAN descriptor". The pSdu is the beacon payload buffer. The `pBufferRoot` pointer contains the data fields pointed to by the other pointers, and is used for freeing only.

WARNING

Free `pBufferRoot` before freeing the indication message, as shown in this example:

```
MEM_BufferFree(pBeaconInd->pBufferRoot);
MEM_BufferFree(pBeaconInd);
```

Otherwise, the MAC memory pools will be exhausted after just a few beacons.

```
// Type: gMlmeBeaconNotifyInd_c,
typedef STRUCT mlmeBeaconNotifyInd_tag
{
    uint8_t                 bsn;
    uint8_t                 pendAddrSpec;
    uint8_t                 sduLength;
    uint8_t*                pAddrList;
    panDescriptor_t*        pPanDescriptor;
    uint8_t*                pSdu;
    void*                   pBufferRoot;
} mlmeBeaconNotifyInd_t;
```

6.2.11. PAN descriptor

The PAN descriptor structure is a common data type used by both the Active/Passive scans and Beacon Notification messages.

NOTE

Use the Link Quality Indication (LQI) as part of the PAN descriptor structure representing an integer value in the range 0x00-0xFF.

In accordance to the IEEE Std. 802.15.4-2006, the `PanDescriptor` structure is:

```
typedef PACKED_STRUCT panDescriptor_tag
{
    uint64_t            coordAddress;
    uint16_t            coordPanId;
    addrModeType_t      coordAddrMode;
    logicalChannelId_t  logicalChannel;
    resultType_t        securityFailure;
    macSuperframeSpec_t superframeSpec;
    bool_t              gtsPermit;
    uint8_t             linkQuality;
    uint8_t             timeStamp[3];
    macSecurityLevel_t  securityLevel;
    keyIdModeType_t     keyIdMode;
    uint64_t            keySource;
    uint8_t             keyIndex;
} panDescriptor_t;
```

In accordance to the IEEE Std. 802.15.4-2011, the `PanDescriptor` structure is:

```
typedef PACKED_STRUCT panDescriptor_tag
{
    uint64_t            coordAddress;
    uint16_t            coordPanId;
    addrModeType_t      coordAddrMode;
    logicalChannelId_t  logicalChannel;
    resultType_t        securityStatus;
    macSuperframeSpec_t superframeSpec;
    bool_t              gtsPermit;
    uint8_t             linkQuality;
    uint8_t             timeStamp[3];
    macSecurityLevel_t  securityLevel;
    keyIdModeType_t     keyIdMode;
    uint64_t            keySource;
    uint8_t             keyIndex;
    uint8_t*            pCodeList;
} panDescriptor_t;
```

The security parameter has the same meaning as the Scan-Request security parameters.

6.3. Start feature

The start feature is not supported on RFD-type devices and derivatives. According to the 802.15.4 Standard (*0, 0*), set the `macShortAddress` PIB attribute to any value different from 0xFFFF before using the start feature, otherwise an error code (gNoShortAddress_c) is returned.

Be default, the system enables `RxOnWhenIdle` when successfully calling the MLME-START.Request primitive. The new (PAN) coordinator starts receiving right away.

If any additional MLME-START.Request primitives are issued, change the superframe configuration after enabling a beaconed network using MLME-START.Request, all information regarding GTS (if GTS is being used) are cleared, and GTS must be set up again by the NWK layer.

6.3.1. Start primitives

This section describes the implementation of the Start-related primitives.

6.3.2. Start-request

Before sending Start-request, set the `macShortAddress` to something other than 0xFFFF.

The Start-request structure is:

```
// Type: gMlmeStartReq_c,
typedef STRUCT mlmeStartReq_tag
{
    uint16_t                panId;
    logicalChannelId_t      logicalChannel;
    channelPageId_t         channelPage;
    uint32_t                startTime;
    uint8_t                 beaconOrder;
    uint8_t                 superframeOrder;
    bool_t                  panCoordinator;
    bool_t                  batteryLifeExtension;
    bool_t                  coordRealignment;
    macSecurityLevel_t      coordRealignSecurityLevel;
    keyIdModeType_t         coordRealignKeyIdMode;
    uint64_t                coordRealignKeySource;
    uint8_t                 coordRealignKeyIndex;
    macSecurityLevel_t      beaconSecurityLevel;
    keyIdModeType_t         beaconKeyIdMode;
    uint64_t                beaconKeySource;
    uint8_t                 beaconKeyIndex;
} mlmeStartReq_t;
```

- startTime—not used.
- coordRealignSecurityLevel—the security level used for coordinator-realignment command frames.
- coordRealignKeyIdMode—identifies the key to be used. This parameter is ignored when coordRealignSecurityLevel parameter is set to 0x00.
- coordRealignKeySource—the originator of the key to be used. This parameter is ignored if the coordRealignKeyIdMode parameter is ignored or set to 0x00.
- coordRealignKeyIndex—the index of the key to be used. This parameter is ignored if the coordRealignKeyIdMode parameter is ignored or set to 0x00.
- beaconSecurityLevel—the security level to be used for beacon frames.
- beaconKeyIdMode—identifies the key to be used. This parameter is ignored if the beaconSecurityLevel parameter is set to 0x00.
- beaconKeySource—the originator of the key to be used. This parameter is ignored if the beaconKeyIdMode parameter is ignored or set to 0x00.
- beaconKeyIndex—the index of the key to be used. This parameter is ignored if the beaconKeyIdMode parameter is ignored or set to 0x00.

6.3.3. Start-confirm

```
// Type: gMlmeStartCnf_c,
typedef STRUCT mlmeStartCnf_tag
{
    resultType_t             status;
} mlmeStartCnf_t;
```

6.4. Sync feature

When executing an MLME-SYNC.Request, the device tries to synchronize with the coordinator beacons. Because there is no MLME-SYNC.Confirm primitive, the best way to detect when synchronization occurs is to set the *macAutoRequest* PIB attribute to FALSE before executing the MLME-SYNC.Request. This forces the MAC to send the MLME-BEACON-NOTIFY.Indication every time a beacon is received. After the first beacon is received, the *macAutoRequest* PIB attribute can be set to TRUE again. If the consecutive beacons (*aMaxLostBeacons*) are lost, the MAC sends the MLME-SYNC-LOSS.Indication.

NOTE

It is very important to set the *macPANId* PIB attribute to a value different from 0xFFFF before executing the MLME-SYNC.Request. If this is not done, the command is ignored by the MAC.

If the *trackBeacon* parameter is TRUE, the MAC attempts to synchronize with the beacon and track all future beacons. If *trackBeacon* is FALSE, the MAC attempts to synchronize with only the next beacon and then goes back to the IDLE state. This also works in combination with the *macAutoRequest* PIB attribute. For example, if *macAutoRequest* is set to TRUE and MLME-SYNC.Request is issued with *trackBeacon* equal to FALSE, the MAC attempts to acquire synchronization and poll out any pending data.

6.5. Synchronization primitives

This section describes implementation of synchronization-related primitives.

6.5.1. Sync-request

```
// Type: gMlmeSyncReq_c,
typedef STRUCT mlmeSyncReq_tag
{
    logicalChannelId_t      logicalChannel;
    channelPageId_t         channelPage;
    bool_t                  trackBeacon;
} mlmeSyncReq_t;
```

6.5.2. Sync-loss-indication

The Sync-loss-indication structure is:

```
// Type: gMlmeSyncLossInd_c,
typedef STRUCT mlmeSyncLossInd_tag
{
    resultType_t            lossReason;
    uint16_t                panId;
    logicalChannelId_t      logicalChannel;
    channelPageId_t         channelPage;
    macSecurityLevel_t      securityLevel;
    keyIdModeType_t         keyIdMode;
    uint64_t                keySource;
    uint8_t                 keyIndex;
} mlmeSyncLossInd_t;
```

- panId—the PAN identifier with which the device lost synchronization, or to which it was realigned.

- logicalChannel—the logical channel on which the device lost synchronization, or to which it was realigned.
- securityLevel—if the primitive was either generated by the device itself following loss of synchronization, or generated by the PAN coordinator upon detection of a PAN ID conflict, the security level is set to 0x00. If the primitive was generated following the reception of either the Coordinator Realignment command or the PAN ID Conflict Notification command; the security level purportedly used by the received MAC frame.
- keyIdMode—if the primitive was either generated by the device itself following loss of synchronization, or generated by the PAN coordinator upon detection of a PAN ID conflict, this parameter is ignored. If the primitive was generated following the reception of either coordinator realignment command or a PAN ID conflict notification command; the mode used to identify the key purportedly used by the originator of the received frame. This parameter is invalid if the securityLevel parameter is set to 0x00.
- keySource—if the primitive was either generated by the device itself following loss of synchronization, or generated by the PAN coordinator upon detection of a PAN ID conflict, this parameter is ignored. If the primitive was generated following the reception of either a Coordinator Realignment command or a PAN ID Conflict Notification command; the originator of the key purportedly used by the originator of the received frame. This parameter is invalid if the keyIdMode parameter is invalid or set to 0x00.
- keyIndex—if the primitive was either generated by the device itself following loss of synchronization, or generated by the PAN coordinator upon detection of a PAN ID conflict, this parameter is ignored. If the primitive was generated following the reception of either a Coordinator Realignment command or a PAN ID Conflict Notification command; the index of the key purportedly used by the originator of the received frame. This parameter is invalid if the keyIdMode parameter is invalid or set to 0x00.

6.6. Association feature

Association is implemented according to the 802.15.4 Standard, but the standard is not explicit on how and when various MAC PIB attributes are updated. Issuing the MLME-ASSOCIATE.Request primitive actually results in these MAC command frames being sent to the coordinator:

- The association request itself.
- The data request that is sent after *aResponseWaitTime* symbols.

See the 802.15.4 Standard for a description of the association procedure. The following attributes are updated when calling MLME-ASSOCIATE.Request.

- macPanId—this attribute is updated as required by the 802.15.4 Standard.
- phyLogicalChannel—this attribute is updated as required by the 802.15.4 Standard.
- macCoordExtendedAddress—this attribute is updated if the extended address of the coordinator is passed as argument. Otherwise, it is not affected.
- macCoordShortAddress—this attribute is updated with the value passed as argument if short addressing mode is used. This is stated in the 802.15.4 Standard. If the extended coordinator address is used in the call, it is not possible to update this attribute, and the short address of the coordinator is unknown. The 802.15.4 Standard does not mention this possibility.

The implementation forces the macCoordShortAddress to 0xFFFE if an extended address is used in the call.

- macShortAddress—the implementation forces this attribute to 0xFFFF before sending the request to the coordinator. This is the default value after a reset. The attribute is updated because it ensures that the data request is sent using long source address. This is the only way to guarantee that the association response is successfully extracted from the coordinator. Setting macShortAddress to 0xFFFF can be considered as a safeguard mechanism. Although this update is not listed in the 802.15.4 Standard, it does not violate the intention of the 802.15.4 Standard.

When these attributes are updated, a MAC command frame containing the association request is sent to the coordinator, and a timer is started upon successful reception. The timer expires after *aResponseWaitTime* symbols.

The timeout value has a different meaning depending on the scenario used:

- Non-beacon-enabled PAN network (macBeaconOrder = 15)—the timeout value is just a simple wait (corresponds to approximately 0.5 s).
- Beacon-enabled PAN network (macBeaconOrder < 15)—the same interpretation as already stated is used if the beacon is not being tracked. If the beacon is being tracked, it implies that the timeout value corresponds to CAP symbols. In this case, the timeout can extend over several superframes.

WARNING

If the beacon is tracked, there are implications that may not be apparent. For example, consider a superframe configuration with macBeaconOrder = 14 and macSuperframeOrder = 0. In this example, the CAP is approximately 900 symbols (depending on the length of the beacon frame). Beacons are transmitted approximately every four minutes. The *aResponseWaitTime* is equal to 30.720 symbols. This implies that the timeout occurs after approximately 34 superframes, which is more than two hours in this example. Always pay attention to the impact of the *aResponseWaitTime* value, because it relates to association requests.

The data request is sent when the timer expires to get the association response from the coordinator, unless the following occurs:

- The association response is “auto-requested” (beacon enabled PAN with beacon tracking enabled and macAutoRequest set to TRUE). The timer is cancelled if the response arrives before the timer expires. The implementation discards all other types of incoming MAC command frames while waiting for the association response.
- Beacon synchronization may be lost on a beacon-enabled PAN. Loss of beacon synchronization implies that the beacon is tracked when the association procedure is initiated. If this happens, the association attempt is aborted with a status error of BEACON LOST (indicated in the MLME-ASSOCIATE.Confirm message). This error code is not listed in the 802.15.4 Standard. The MLME-SYNC-LOSS.Indication message is also generated (as expected when synchronization is lost).

Once the data request has been sent (if any), the code is ready to process any incoming MAC command frames (the expected being the MLME-ASSOCIATE.Response packet of course). These attributes are updated if the associate response frame is received and the status indicates a successful association:

- macShortAddress—this attribute is updated with the allocated short address.
- macCoordExtendedAddress—the source address is extracted from the MAC command frame header and stored in this attribute.
- macCoordShortAddress—this attribute is not updated, although it is mentioned in the 802.15.4 Standard. The short address of the coordinator is not present in the response.

The MLME-COMM-STATUS.Indication message is generated on the coordinator when the response is extracted by the device.

6.7. Association primitives

This section describes the implementation of the Association-related primitives.

6.7.1. Associate-request

Before sending the Associate-request primitive, the 802.15.4 Standard states that a Reset-request must be sent, and the Active or Passive scan is performed.

The Associate-request structure is:

```
// Type: gMlmeAssociateReq_c,
typedef STRUCT mlmeAssociateReq_tag
{
    uint64_t                coordAddress;
    uint16_t                coordPanId;
    addrModeType_t          coordAddrMode;
    logicalChannelId_t      logicalChannel;
    macSecurityLevel_t      securityLevel;
    keyIdModeType_t         keyIdMode;
    uint64_t                keySource;
    uint8_t                 keyIndex;
    macCapabilityInfo_t     capabilityInfo;
    channelPageId_t         channelPage;
} mlmeAssociateReq_t;
```

The security parameter has the same meaning as the Scan-request security parameters.

6.7.2. Associate-response

The Association-response structure is:

```
// Type: gMlmeAssociateRes_c,
typedef STRUCT mlmeAssociateRes_tag
{
    uint64_t                deviceAddress;
    uint16_t                assocShortAddress;
    macSecurityLevel_t      securityLevel;
    keyIdModeType_t         keyIdMode;
    uint64_t                keySource;
    uint8_t                 keyIndex;
    resultType_t            status;
} mlmeAssociateRes_t;
```

The security parameter has the same meaning as the Scan-request security parameters.

6.7.3. Associate-indication

The Associate-indication structure is:

```
// Type: gMlmeAssociateInd_c,
typedef STRUCT mlmeAssociateInd_tag
{
    uint64_t                deviceAddress;
    macSecurityLevel_t      securityLevel;
    keyIdModeType_t         keyIdMode;
    uint64_t                keySource;
    uint8_t                 keyIndex;
    macCapabilityInfo_t     capabilityInfo;
} mlmeAssociateInd_t;
```

- securityLevel—the security level purportedly used by the received MAC command frame.
- keyIdMode—the mode used to identify the key purportedly used by the originator of the received frame. This parameter is invalid if the securityLevel parameter is set to 0x00.
- keySource—the originator of the key purportedly used by the originator of the received frame. This parameter is invalid if the keyIdMode parameter is invalid or set to 0x00.
- keyIndex—the index of the key purportedly used by the originator of the received frame. This parameter is invalid if the keyIdMode parameter is invalid or set to 0x00.

6.7.4. Associate-confirm

The Associate-confirm structure is:

```
// Type: gMlmeAssociateCnf_c,
typedef STRUCT mlmeAssociateCnf_tag
{
    uint16_t                assocShortAddress;
    resultType_t            status;
    macSecurityLevel_t      securityLevel;
    keyIdModeType_t         keyIdMode;
    uint64_t                keySource;
    uint8_t                 keyIndex;
} mlmeAssociateCnf_t;
```

- securityLevel—if the primitive is generated following failed outgoing processing of an association request command—the security level to be used. If the primitive is generated following receipt of an association response commands—the security level purportedly used by the received frame.
- keyIdMode—if the primitive is generated following failed outgoing processing of an association request command—the mode used to identify the key to be used. This parameter is ignored if the securitylevel parameter is set to 0x00. If the primitive is generated following the receipt of association response command—the mode used to identify the key purportedly used by the originator of the received frame. This parameter is invalid if the securityLevel parameter is set to 0x00.
- keySource—if the primitive is generated following failed outgoing processing of an association request command—the originator of the key to be used. This parameter is ignored if the keyIdMode parameter is ignored or set to 0x00. If the primitive is generated following a receipt of association response command—the originator of the key purportedly used by the originator of the received frame. This parameter is invalid if the keyIdMode parameter is invalid or set to 0x00.
- keyIndex—if the primitive is generated following failed outgoing processing of an association request command—the index of the key to be used. This parameter is ignored if the keyIdMode parameter is ignored or set to 0x00. If the primitive is generated following a receipt of association response command—the index of the key purportedly used by the originator of the received frame. This parameter is invalid if the keyIdMode parameter is invalid or set to 0x00.

6.7.5. Associate example

Example 11. Sending associate request

```
/* Static information about the PAN Coordinator */
static panDescriptor_t mCoordInfo;

uint8_t App_SendAssociateRequest_Example( void )
{
    mlmeMessage_t *pMsg;

    /* Allocate a message for the MLME message. */
    pMsg = MEM_BufferAlloc( sizeof(mlmeMessage_t) );
    if( pMsg != NULL )
    {
        /* This is a MLME-ASSOCIATE.Req command. */
        pMsg->msgType = gMlmeAssociateReq_c;

        /* Create the Associate request message data. */

        /* Use the coordinator info from a previous Active Scan. */
        pMsg->msgData.associateReq.coordAddress     = mCoordInfo.coordAddress;
        pMsg->msgData.associateReq.coordPanId       = mCoordInfo.coordPanId;
        pMsg->msgData.associateReq.coordAddrMode    = mCoordInfo.coordAddrMode;
        pMsg->msgData.associateReq.logicalChannel   = mCoordInfo.logicalChannel;
        pMsg->msgData.associateReq.securityLevel    = gMacSecurityNone_c;

        /* We want the coordinator to assign a short address to us. */
        pMsg->msgData.associateReq.capabilityInfo   = gCapInfoAllocAddr_c;

        /* Send the Associate Request to the MLME. */
        if( NWK_MLME_SapHandler( pMsg, mMacInstance ) == gSuccess_c )
        {
            return errorNoError;
        }
        else
        {
            /* One or more parameters in the message were invalid. */
            return errorInvalidParameter;
        }
    }
    else
    {
        /* Allocation of a message buffer failed. */
        return errorAllocFailed;
    }
}
```

6.8. Disassociation feature

Disassociation is less complex than association, but there is an issue in the 802.15.4 Standard that makes disassociation from a coordinator difficult in a non-beacon-enabled PAN network. Pay attention when disassociating from a coordinator in a non-beacon network.

Disassociation from a device—the MLME-DISASSOCIATE.Request is sent to the remote device where it results in MLME-DISASSOCIATE.Indication message.

The 802.15.4 Standard states that a device with a valid short address supplies this address as a source address in the MAC header of the data request. However, the coordinator must queue the packet using the extended address of the device. The result is that the packet cannot be extracted from the coordinator because the short address cannot be matched against the long address, so the MLME-DISASSOCIATE.Request is queued in the indirect queue. It resides there until it is polled by the remote device (or the transaction expires).

It is not possible to disassociate from a coordinator in a non-beacon-enabled PAN if the device has a valid short address (address < 0xFFFE)

This limitation does not exist on a beacon-enabled PAN where macAutoRequest = TRUE, because the auto-request poll packet is sent with a source address equal to the one indicated in the beacon frame pending address list.

A workaround exists for all other scenarios. That is, the device can temporarily set its macShortAddress to 0xFFFE or 0xFFFF if it wishes to poll for packets queued using the device’s extended address.

6.9. Disassociation primitives

This section describes the implementation of the Disassociation-related primitives.

6.9.1. Disassociate-request

The Disassociate-request structure is:

```
// Type: gMlmeDisassociateReq_c,
typedef STRUCT mlmeDisassociateReq_tag
{
    uint64_t                deviceAddress;
    uint16_t                devicePanId;
    addrModeType_t           deviceAddrMode;
    macDisassociateReason_t disassociateReason;
    bool_t                  txIndirect;
    macSecurityLevel_t      securityLevel;
    keyIdModeType_t         keyIdMode;
    uint64_t                keySource;
    uint8_t                 keyIndex;
} mlmeDisassociateReq_t;
```

- devicePanId—the PAN identifier of the device to which to send the disassociation notification command.
- deviceAddrMode—the addressing mode of the device to which to send the disassociation notification command.
- txIndirect—this is TRUE if the disassociation notification command is to be sent indirectly.

Security parameters have the same meaning as Scan-Request security parameters.

6.9.2. Disassociate-indication

The Disassociate-indication structure is:

```
// Type: gMlmeDisassociateInd_c,
typedef STRUCT mlmeDisassociateInd_tag
{
    uint64_t                deviceAddress;
    macDisassociateReason_t disassociateReason;
    macSecurityLevel_t      securityLevel;
    keyIdModeType_t         keyIdMode;
    uint64_t                keySource;
    uint8_t                 keyIndex;
} mlmeDisassociateInd_t;
```

- securityLevel—the security level purportedly used by the received MAC command frame.
- keyIdMode—identifies the key purportedly used by the originator of the received frame. This parameter is invalid if the securityLevel parameter is set to 0x00.
- keySource—the originator of the key purportedly used by the originator of the received frame. This parameter is invalid if the keyIdMode parameter is invalid or set to 0x00.
- keyIndex—the index of the key purportedly used by the originator of the received frame. This parameter is invalid if the keyIdMode parameter is invalid or set to 0x00.

6.9.3. Disassociate-confirm

The Disassociate-confirm structure is:

```
// Type: gMlmeDisassociateCnf_c,
typedef STRUCT mlmeDisassociateCnf_tag
{
    uint64_t                deviceAddress;
    uint16_t                devicePanId;
    addrModeType_t          deviceAddrMode;
    resultType_t            status;
} mlmeDisassociateCnf_t;
```

- deviceAddress—the address of the device that either requested disassociation, or is instructed to disassociate by its coordinator.
- devicePanId—the PAN identifier of the device that either requested disassociation or is instructed to disassociate by its coordinator.
- deviceAddrMode—the addressing mode of the device that either requested disassociation or is instructed to disassociate by its coordinator.

6.10. Data feature

The data feature includes the service provided by the MCPS-DATA.Confirm/Indication and MLME-POLL.Request/Confirm primitives. Whenever these primitives are in use, so are one or more large data buffers. The large data buffers are mainly used for holding Tx or Rx packets, and they are limited to a specific number for each Device Type.

Each time the MCPS-DATA.Confirm or MLME-POLL.Request primitives are executed, one large buffer is used. Even though not directly supported by the 802.15.4 Standard, it is possible to execute the MLME-POLL.Request while another MLME-POLL.Request is pending in the MAC.

The MAC reserves a buffer for general receive and for transmitting beacons, unless it is running in the non-beacon mode as a device. This means that it is safe for the application to allocate data buffers using the *MEM_BufferAlloc()* function until receiving NULL, indicating that no buffers are available.

6.11. Data primitives

This section describes the implementation of the Data-related primitives.

6.11.1. Data-request

The Data-request message structure has an embedded data field. The total size of the message is sizeof(nwkToMcpsMessage_t) + gPhyMaxDataLength_c. The data field is simply addressed with mcpsDataReq->pMsdu, and can contain more than one byte.

The Data-request structure is:

```
// Type: gMcpsDataReq_c,
typedef STRUCT mcpsDataReq_tag
{
    uint16_t                srcPanId; /* Not in Spec */
    uint64_t                srcAddr; /* Not in Spec */
    addrModeType_t          srcAddrMode;
    addrModeType_t          dstAddrMode;
    uint16_t                dstPanId;
    uint64_t                dstAddr;
    uint16_t                msduLength;
    uint8_t                 msduHandle;
    macTxOptions_t          txOptions;
    macSecurityLevel_t      securityLevel;
    keyIdModeType_t         keyIdMode;
    uint64_t                keySource;
    uint8_t                 keyIndex;
    uint8_t                 *pMsdu;
} mcpsDataReq_t;
```

The security parameters have the same meaning as the Scan-request security parameters.

6.11.2. Data-confirm

```
// Type: gMcpsDataCnf_c,
typedef STRUCT mcpsDataCnf_tag
{
    uint8_t                 msduHandle;
    resultType_t            status;
    uint32_t                timestamp;
} mcpsDataCnf_t;
```

6.11.3. Data-indication

The Data-indication structure has an embedded data field. The total size of the message is sizeof(mcpsToNwkMessage_t) + gMacMaxMACPayloadSize_c. The data field is simply addressed with mcpsDataInd->pMsdu, and can contain more than one byte.

NOTE

Link Quality Indication (LQI) is used as part of the Data-indication structure representing an integer value in the range of 0x00-0xFF.

The Data-indication structure is the following:

```
// Type: gMcpsDataInd_c,
typedef STRUCT mcpsDataInd_tag
{
    uint64_t                dstAddr;
    uint16_t                dstPanId;
    addrModeType_t          dstAddrMode;
    uint64_t                srcAddr;
    uint16_t                srcPanId;
    addrModeType_t          srcAddrMode;
    uint16_t                msduLength;
    uint8_t                 mpduLinkQuality;
    uint8_t                 dsn;
    uint32_t                timestamp;
    macSecurityLevel_t      securityLevel;
    keyIdModeType_t         keyIdMode;
    uint64_t                keySource;
    uint8_t                 keyIndex;
    uint8_t                 *pMsdu;
} mcpsDataInd_t;
```

- dsn—the data sequence number of the received frame.
- timestamp—the time (in symbols) in which data were received.
- securityLevel—the security level purportedly used by the received data frame.
- keyIdMod—the mode used to identify the key purportedly used by the originator of the received frame. This parameter is invalid if the securityLevel parameter is set to 0x00.
- keySource—the originator of the key purportedly used by the originator of the received frame. This parameter is invalid if the keyIdMode parameter is invalid or set to 0x00.
- keyIndex—the index of the key purportedly used by the originator of the received frame. This parameter is invalid if the keyIdMode parameter is invalid or set to 0x00.

6.11.4. Poll-request

The Poll-request structure is:

```
// Type: gMlmePollReq_c,
typedef STRUCT mlmePollReq_tag
{
    addrModeType_t          coordAddrMode;
    uint16_t                coordPanId;
    uint64_t                coordAddress;
    macSecurityLevel_t      securityLevel;
    keyIdModeType_t         keyIdMode;
    uint64_t                keySource;
    uint8_t                 keyIndex;
} mlmePollReq_t;
```

The security parameters have the same meaning as the Scan-request security parameters.

6.11.5. Poll-confirm

```
// Type: gMlmePollCnf_c,
typedef STRUCT mlmePollCnf_tag
{
    resultType_t            status;
} mlmePollCnf_t;
```

6.11.6. Poll notify-indication

```
// Type: gMlmePollNotifyInd_c,
typedef STRUCT mlmePollNotifyInd_tag
{
    addrModeType_t          srcAddrMode;
    uint64_t                srcAddr;
    uint16_t                srcPanId;
} mlmePollNotifyInd_t;
```

6.11.7. Communications status-indication

The Communications status-indication structure is:

```
// Type: gMlmeCommStatusInd_c,
typedef STRUCT mlmeCommStatusInd_tag
{
    uint64_t                srcAddress;
    uint16_t                panId;
    addrModeType_t          srcAddrMode;
    uint64_t                destAddress;
    addrModeType_t          destAddrMode;
    resultType_t            status;
    macSecurityLevel_t      securityLevel;
    keyIdModeType_t         keyIdMode;
    uint64_t                keySource;
    uint8_t                 keyIndex;
} mlmeCommStatusInd_t;
```

- securityLevel—if the primitive is generated after a transmission instigated through a response primitive—the security level to be used. If the primitive is generated on receipt of a frame that generates an error in its security processing—the security level purportedly used by the received frame.
- keyIdMode—if the primitive is generated after a transmission instigated through a response primitive—the mode used to identify the key to be used. This parameter is ignored if the securityLevel parameter is set to 0x00. If the primitive is generated on receipt of a frame that generates an error in its security processing—the mode used to identify the key purportedly used by the originator of the received frame. This parameter is invalid if the securityLevel parameter is set to 0x00.
- keySource—if the primitive is generated after a transmission instigated through a response primitive—the originator of the key to be used. This parameter is ignored if the keyIdMode parameter is ignored or set to 0x00. If the primitive is generated on receipt of a frame that generates an error in its security processing—the originator of the key purportedly used by the originator of the received frame. This parameter is invalid if the keyIdMode parameter is invalid or set to 0x00.
- keyIndex—if the primitive is generated after a transmission instigated through a response primitive—the index of the key to be used. This parameter is ignored if the keyIdMode parameter is ignored or set to 0x00. If the primitive is generated on receipt of a frame that generates an error in its security processing—the index of the key purportedly used by the originator of the received frame. This parameter is invalid if the keyIdMode parameter is invalid or set to 0x00.

6.11.8. Data example

The following is an example of the NWK sending MCPS-DATA.Request to the MCPS. It shows how to properly allocate a message buffer, and add data to the pMsdu parameter.

Example 12. Sending MCPS-DATA.Request to MCPS

```
#define mPredefinedMsg_c                    "Predefined message!\n\r"
#define mPredefinedMsgLen_c                 (22)
static uint8_t maGeneralMsg[mPredefinedMsgLen_c] = mPredefinedMsg_c;

/* Static information about the PAN Coordinator */
static panDescriptor_t mCoordInfo;

uint8_t App_SendDataRequest_Example(instanceId_t instanceId)
{
    nwkToMcpsMessage_t *pMsg;
    pMsg = MEM_BufferAlloc(sizeof(nwkToMcpsMessage_t) + gPhyMaxDataLength_c)
    if( pMsg != NULL )
    {
        pMsg->msgType = gMcpsDataReq_c;
        /* initialize pointer to Msdu */
        pMsg->msgData.dataReq.pMsdu = (uint8_t*)(&(pMsg->msgData.dataReq.pMsdu)) +
sizeof(uint8_t*);
        FLib_MemCpy(pMsg->msgData.dataReq.pMsdu, (uint8_t*)maGeneralMsg,
mPredefinedMsgLen_c);
        pMsg->msgData.dataReq.dstAddr = mCoordInfo.coordAddress;
        pMsg->msgData.dataReq.srcAddr = mAddress;
        pMsg->msgData.dataReq.dstPanId = mCoordInfo.coordPanId;
        pMsg->msgData.dataReq.srcPanId = mCoordInfo.coordPanId;
        pMsg->msgData.dataReq.dstAddrMode = mCoordInfo.coordAddrMode;
        pMsg->msgData.dataReq.srcAddrMode = gAddrModeShortAddress_c;
        pMsg->msgData.dataReq.msduLength = mPredefinedMsgLen_c;

        /* Request MAC level acknowledgement of the data packet */
        pMsg->msgData.dataReq.txOptions = gMacTxOptionsAck_c;
        pMsg->msgData.dataReq.msduHandle = mMsduHandle++;
        pMsg->msgData.dataReq.securityLevel = gMacSecurityNone_c;

        /* Send the Data Request to the MCPS */
       (void)NWK_MCPS_SapHandler(pMsg, instanceId);
    }
}
```

The following is an example of NWK receiving MCPS-DATA.Indication from the MCPS. It shows how to implement the MCPS to NWK SAP handler using the application/NWK programmer.

Example 13. NWK receiving MCPS-DATA.Indication from MCPS

```
static void MCPS_NWK_SapHandler(mcpsToNwkMessage_t *pMsg, instanceId_t instanceId)
{
    uint8_t myBuffer[100];

    switch(pMsg->msgType)
    {
        case gMcpsDataInd_c:
            // Handle the incoming data frame
            for(i=0; i<pMsg->msgData.dataInd.msduLength; i++)
            {
                myBuffer[i] = pMsg->msgData.dataInd.pMsdu[i];
            }
            break;
        case gMcpsDataCnf_c:
            // The MCPS-DATA.Request has completed. Check status
            // parameter to see if the transmission was successful.
            break;
        case gMcpsPurgeCnf_c:
            // The MCPS-PURGE.Request has completed.
            break;
    }
    MEM_BufferFree(pMsg); // Free message ASAP.
    return;
}
```

6.12. Purge feature

The purge feature enables the next higher layer to purge a data packet (MSDU) stored in the MAC until it is sent. This means that if an MCPS-DATA.Request primitive is initiated with msduHandle, it is possible to purge the MSDU with the given msduHandle (if it is not sent). Initiating the MCPS-PURGE.Request primitive and specifying the msduHandle parameter completes the task.
A MCPS-PURGE.Confirm primitive is generated in response to the MCPS-PURGE.Request primitive with the status of SUCCESS if an MSDU matching the given handle is found, or with the status of INVALID_HANDLE (if an MSDU matching the given handle is not found).

6.13. Purge primitives

This section describes the implementation of the Purge-related primitives.

6.13.1. Purge-request

```
// Type: gMcpsPurgeReq_c,
typedef STRUCT mcpsPurgeReq_tag
{
    uint8_t                 msduHandle;
} mcpsPurgeReq_t;
```

6.13.2. Purge-confirm

```
// Type: gMcpsPurgeCnf_c,
typedef STRUCT mcpsPurgeCnf_tag
{
    uint8_t                 msduHandle;
    resultType_t            status;
} mcpsPurgeCnf_t;
```

6.14. Rx Enable feature

The Rx Enable feature enables the network layer to enable the receiver at a given time. The feature is implemented according to the 802.15.4 Standard.

6.14.1. RX-enable-request

In accordance with IEEE Std. 802.15.4-2006, the Rx-Enable-Request structure is:

```
// Type: gMlmeRxEnableReq_c,
typedef STRUCT mlmeRxEnableReq_tag
{
    bool_t                  deferPermit;
    uint32_t                rxOnTime;
    uint32_t                rxOnDuration;
} mlmeRxEnableReq_t;
```

In accordance to the IEEE Std. 802.15.4-2011, the Rx-Enable-Request structure is:

```
// Type: gMlmeRxEnableReq_c,
typedef STRUCT mlmeRxEnableReq_tag
{
    bool_t                  deferPermit;
    uint32_t                rxOnTime;
    uint32_t                rxOnDuration;
    macRangingRxControl_t   rangingRxControl;
} mlmeRxEnableReq_t;
```

6.14.2. RX-enable-confirm

```
// Type: gMlmeRxEnableCnf_c,
typedef STRUCT mlmeRxEnableCnf_tag
{
    resultType_t            status;
} mlmeRxEnableCnf_t;
```

6.15. Guaranteed Time Slots (GTS) feature

The GTS feature enables a device to reserve a certain bandwidth. The GTS slot is always unidirectional, and it is always requested by the device.

6.15.1. GTS as a device

It does not make sense for a device to allocate more than one Rx slot and one Tx slot (although it is possible to do so) because it is impossible for a device to differentiate two Tx slots of the same length. Analysis yields these results:

- The MCPS-DATA.Request does not support any method of switching between Tx slots.
- If the PAN coordinator deallocates or realigns one of the Tx slots, it is not possible to tell which of the slots were affected.

Freescale recommends is that a device should never allocate more than one GTS slot in each direction.

See the 802.15.4 Standard for more information. Allocating a GTS slot or deallocating a GTS slot is implemented according to the standard. In either case, the MLME-GTS.Request primitive is used.

- Allocating—an allocation attempt is initiated by sending the GTS request to the PAN coordinator. The device then looks for a GTS descriptor that matches the requested characteristics in the beacon frames received. Once found, it is possible to perform GTS transfers.
- Deallocating—deallocation is similar to allocation. The local GTS “context” is marked as invalid before the request is actually sent to the PAN coordinator. Any packets that are queued for GTS transmission are completed with status INVALID_GTS at this point. The GTS deallocation request is then sent to the PAN coordinator. There is no guarantee that the PAN coordinator receives the request (it can fail with status NO_ACK or CHANNEL_ACCESS_FAILURE). This is not critical because the PAN coordinator must implement mechanisms to detect “stale” GTS slots.

NOTE

GTS processing is MCU-expensive and cannot be completed in IRQ context.

The following steps describe the procedure for handling GTS:

1. A beacon frame is received.
2. Time-critical beacon frame processing is performed. This includes calculating various superframe timing parameters, such as the expected end time of the CAP, and the expected time of the next beacon frame arrival.
3. The GTS field of the beacon frame is preprocessed. Preprocessing consists of only one thing: Check if the device's short address is present in the list. An internal flag is raised if this is true.
4. The beacon frame is then queued for further processing by higher-layer software (the MLME).
5. This completes the beacon processing in IRQ context.

The MLME asynchronously performs further processing of the beacon frame. This includes generating MLME-BEACON.Indication messages for the NWK layer. GTS processing is performed if the internal flag is raised. This includes processing all GTS descriptors that match the short address of the device. The following actions are performed.

1. A new internal GTS context is allocated if a GTS allocation request was pending and the current GTS descriptor matches the requested characteristics.
2. An existing GTS slot is realigned by the PAN coordinator (that is, a new start slot is defined). The proper internal GTS context is updated.
3. An existing GTS slot is deallocated by the PAN coordinator (indicated by a start slot of 0). The proper internal GTS context is deallocated. Queued data packets are completed with status INVALID_GTS (if applicable).
4. Timing parameters for the entire CFP are then calculated. This includes calculating the start and end times for all allocated GTS slots. The times are adjusted according to internal setup requirements and clock drift.
5. The internal flag is cleared.

NOTE

These steps are important because the entire CFP of a superframe is skipped if there is no active GTS at the beginning of the CFP. This is needed because the CFP timing parameters are not yet in place. It is also important to call the *Mac_Task()* in a timely manner.

6.15.2. GTS as PAN coordinator

The PAN coordinator always accepts incoming GTS requests and it always allocates the requested GTS slot if the minimum-length CAP is maintained. A GTS slot can occupy one or more superframe slots (assuming that sufficient superframe slots are free). A GTS request is denied if the PAN coordinator cannot allocate the requested GTS slot.

NOTE

GTS processing is rather intensive and cannot be completed in IRQ context. The following steps describe the procedure for this software implementation.

1. A GTS request is received in the CAP.
2. An internal flag is raised.
3. The MAC command frame is queued for further processing by higher-layer software (the MLME).
4. This completes the GTS processing in IRQ context.

As previously stated, the MLME asynchronously performs further processing of the GTS request. This includes these actions:

1. An internal GTS context is allocated (if the GTS request specified an allocation request), or an existing context is deallocated (if the GTS request specified a deallocation request).
2. All GTS slots are realigned if deallocation created “gaps” in the CFP.
3. Timing parameters for the entire CFP are calculated. This includes calculating the start and end times for all allocated GTS slots. The times are adjusted according to internal setup requirements and clock drift.
4. The internal flag is cleared.
5. The MLME-GTS.Indication message is generated (if applicable).

NOTE

These steps are important because the entire CFP of a superframe is skipped if there is no active GTS at the beginning of the CFP. This is required because the CFP timing parameters are not yet in place. It is therefore important that you call the *Mac_Task()* in a timely manner.

Be aware that all existing GTS slots (if any) are deallocated immediately if the superframe configuration changes (*macBeaconOrder* or *macSuperframeOrder* are changed).

NOTE

Issuing the MLME-START.Request primitive updates the two mentioned PIB attributes. There is no indication in the beacon frame to indicate this. The device must assume that the GTS slots are invalidated if the superframe configuration changes.

GTS expiration is implemented according to the 802.15.4 Standard. The PAN coordinator deallocates stale slots automatically. The 802.15.4 Standard does not specify how to expire a GTS slot where data is sent unacknowledged. This implies that the PAN coordinator will not receive any acknowledgement frames. The implementation in this case deallocates the GTS slot if no data has been transmitted in the slot for the specified number of superframes. For example, “counting” is based on Tx packets and not Rx acknowledgements.

6.15.3. Miscellaneous items

The following items are valid for both a device and a PAN coordinator. It is possible (but not recommended) to allocate a GTS slot with a length of 1 at macSuperframeOrder = 0. This GTS slot only contains 60 symbols (30 bytes). Setup time and overhead for PHY and MAC headers is at least 21 bytes, so it is not possible to send or receive any data. A GTS slot must have a length of 2 at macSuperframeOrder of 0 (corresponding to 120 symbols). All other superframe orders support GTS slots with a length of 1. You can skip CFP due to GTS maintenance. Although this hazard exists, it has minimal effect because GTS slots are rarely allocated and deallocated, except for feature setup and termination.

6.15.4. GTS primitives

6.15.4.1. GTS-request

The GTS-request structure is:

```
// Type: gMlmeGtsReq_c,
typedef STRUCT mlmeGtsReq_tag
{
    gtsCharacteristics_t    gtsCharacteristics;
    macSecurityLevel_t      securityLevel;
    keyIdModeType_t         keyIdMode;
    uint64_t                keySource;
    uint8_t                 keyIndex;
} mlmeGtsReq_t;
```

The security parameters have the same meaning as Scan-request security parameters.

6.15.4.2. GTS-confirm

```
// Type: gMlmeGtsCnf_c,
typedef STRUCT mlmeGtsCnf_tag
{
    resultType_t            status;
    gtsCharacteristics_t    gtsCharacteristics;
} mlmeGtsCnf_t;
```

6.15.4.3. GTS-indication

The GTS-indication structure is:

```
// Type: gMlmeGtsInd_c,
typedef STRUCT mlmeGtsInd_tag
{
    uint16_t                deviceAddress;
    gtsCharacteristics_t    gtsCharacteristics;
    macSecurityLevel_t      securityLevel;
    keyIdModeType_t         keyIdMode;
    uint64_t                keySource;
    uint8_t                 keyIndex;
} mlmeGtsInd_t;
```

- securityLevel—if the primitive is generated when GTS deallocation is initiated by the PAN coordinator itself, the security level to be used is set to 0x00. If the primitive is generated whenever a GTS is allocated or deallocated after the reception of a GTS request command—the security level purportedly used by the received MAC command frame.
- keyIdMode—if the primitive is generated when GTS deallocation is initiated by the PAN coordinator itself, this parameter is ignored. If the primitive is generated whenever a GTS is allocated or deallocated after the reception of a GTS request command—the mode used to identify the key purportedly used by the originator of the received frame. This parameter is invalid if the securityLevel parameter is set to 0x00.
- keySource—if the primitive is generated when GTS deallocation is initiated by the PAN coordinator itself, this parameter is ignored. If the primitive is generated whenever a GTS is allocated or deallocated after the reception of a GTS request command—the originator of the key purportedly used by the originator of the received frame. This parameter is invalid if the keyIdMode parameter is invalid or set to 0x00.
- keyIndex—if the primitive is generated when GTS deallocation is initiated by the PAN coordinator itself, this parameter is ignored. If the primitive is generated whenever a GTS is allocated or deallocated after the reception of a GTS request command—the index of the key purportedly used by the originator of the received frame. This parameter is invalid if the keyIdMode parameter is invalid or set to 0x00.

6.16. Security

The MAC Security functionality is implemented as described in the 802.15.4 Standard (*0, 0*) and in the *ZigBee Security Services Specification*. In cases where a different approach is used between the two, Freescale follows the *ZigBee Security Services Specification*. The CCM security levels are used in place of the security suites described in the 802.15.4 Standard. Secured packets are longer than corresponding non-secured packets when transmitted over the air. Apart from increased power consumption and lower maximum throughput, this results in MCPS-DATA.Request delivering gFrameTooLong_c error code if the resulting packet is longer than 127 bytes. The maximum msduLength for secured packets depends on the security level, source and destination addressing modes, and whether the source and destination PAN ID are the same. This table shows the overhead added for each security level.

Security level overhead

Level	Name	Encrypted	Integrity Check (length)	Packet Length Overhead
0x00	N/A	No	0 (no check)	0
0x01	MIC-32	No	4	9
0x02	MIC-64	No	8	13
0x03	MIC-128	No	16	21
0x04	ENC	Yes	0 (no check)	5
0x05	ENC-MIC-32	Yes	4	9
0x06	ENC-MIC-64	Yes	8	13
0x07	ENC-MIC-128	Yes	16	21

6.16.1. Security PIB attributes

The number of entries for *macKeyTable, KeyIdLookupList, KeyDeviceList, KeyUsageList, macDeviceTable,* and *macSecurityLevelTable* are allocated statically. For each table, the number of entries is defined in *AppToMacPhyConfig.h* file: gNumKeyTableEntries_c, gNumKeyIdLookupEntries_c, gNumKeyDeviceListEntries_c, gNumKeyUsageListEntries_c, gNumDeviceTableEntries_c, gNumSecurityLevelTableEntries_c. Each *macKeyTable* entry has the number of KeyIdLookupList, KeyDeviceList and KeyUsageList entries defined by the gNumKeyIdLookupEntries_c, gNumKeyDeviceListEntries_c, gNumKeyUsageListEntries_c constants.

The PIB attributes for the actual tables and their numbers of entries (e.g., gMPibKeyTable_c, gMPibKeyTableEntries_c, gMPibKeyIdLookupList_c, gMPibKeyIdLookupListEntries_c, and so on) are read-only. They are set at compile time and cannot be dynamically changed. There is no need to read the entire table for reading/writing entries into the security table or subtable. The corresponding index PIB attributes must be set (gMPibiKeyTableCrtEntry_c, gMPibiDeviceTableCrtEntry_c, gMPibiSecurityLevelTableCrtEntry_c, gMPibiKeyIdLookuplistCrtEntry_c, gMPibiKeyDeviceListCrtEntry_c, gMPibiKeyUsageListCrtEntry_c), and individual entries can be read/written using a specific PIB attribute.

For example, when the gMPibKeyUsageFrameType_c (0x86) PIB attribute is read/written, the actual attribute is determined by the table index PIB attributes: macKeyTable[gMPibiKeyTableCrtEntry_c].KeyUsageList[gMPibiKeyUsageListCrtEntry_c].FrameTyp e.

In the current implementation, it is not possible to read/write entire security tables with a single GetPIB/SetPIB operation. The entries in the tables/subtables must be accessed individually.

6.16.2. Security library

The basic building blocks used by the 802.15.4 security standard are available in the security library. It is not necessary to utilize these functions for any network or application layers for 802.15.4 nodes to work with security. They are provided to be used in cases where any network or application layers may need them for additional security on these layers (hash functions for key generation/exchange protocols or for separate CCM security on the packets).

NOTE

These functions are not reentrant, neither individually or mutually. This requires all calls to these functions to happen from execution contexts that do not interleave, one of these being the execution context from which the MAC main function is called. The reason is that these functions modify some global variables without using mutual exclusion mechanism.

6.16.2.1. Advanced Encryption Standard (AES)

These libraries perform AES-128 cryptographic operation on a data set. Kinetis platforms use *lib_crypto_IAR_M4.a* or *lib_crypto_IAR_M0.a*, which contain software-implemented AES128, SHA1, and SHA256 algorithms.

Interface assumptions

- All input/outputs are 16 bytes (128-bit).
- You can point the pOutput pointer to the same memory as either the Data or Key (if needed).

NOTE

The function is not reentrant. It is not reentrant also with other functions calling this function (like AES_128_CCM).

```
void AES_128_Encrypt
  (
  const uint8_t *pInput, // IN:  Pointer to the location of the 16-byte plain text
  const uint8_t *pKey,  // IN:  Pointer to the location of the 128-bit key
  uint8_t *pOutput  // OUT: Pointer to the location to store the 16-byte ciphered output
  );
```

6.16.2.2. Counter with CBC-MAC (CCM*)

CCM* mode is a mode of operation for cryptographic block ciphers. It is an authenticated encryption algorithm designed to provide both authentication and privacy. CCM* mode is only defined for 128-bit block ciphers. CCM* (as defined for ZigBee technology) offers encryption-only and integrity-only capabilities.

Interface assumptions

Header, *Message*, and *Integrity code* must be located in memory as they appear in the packet. That is, as a concatenated string in that order. For security levels that employ encryption, the message is encrypted in place (overwriting Message).

NOTE

The function is not reentrant.

The function returns the status of the operation (always ok = 0 for encoding).

```
resultType_t AES_128_CCM_Star
(
uint8_t* pHeader,             // IN/OUT: Start of the data to perform CCM* on.
uint8_t headerLength,         // IN: Length of the header field.
uint8_t messageLength,        // IN: Length of data field.
const uint8_t key[16],        // IN: 128 bit key
const uint8_t nonce[13],      // IN: 802.15.4/Zigbee technology-specific nonce.
const uint8_t securityLevel, // IN: 802.15.4-specific Security Level.
ccmDirection_t direction     // IN: Direction of CCM: gCcmEncode_c, gCcmDecode_c
);
```

6.17. Multiple PAN feature

This section is preliminary, and the feature is available for the MKW2x families and for the MCR20A-based solutions, for both MAC2006 and MAC2011 standards.

The Multiple PAN Manager (MPM) takes advantage of MKW2xD's and the MCR20A's DualPAN hardware support to accelerate the on-the-fly channel switch.

To operate on multiple different PANs, the MAC's data (all internal context and PIB table) must be multiplied accordingly by setting the *gMpmMaxPANs_c* and *gMacInstancesCnt_c* defines to a value greater than 1 to enable Multiple PAN Manager, and include one of the following MAC libraries (depending on the MAC version used):

- 802.15.4_mac_06begts_cm0_iar.a
- 802.15.4_mac_06begts_cm4_iar.a
- 802.15.4_mac_06rfd_cm0_iar.a
- 802.15.4_mac_06rfd_cm4_iar.a
- 802.15.4_mac_06_cm0_iar.a
- 802.15.4_mac_06_cm4_iar.a
- 802.15.4_mac_11_cm0_iar.a
- 802.15.4_mac_11_cm4_iar.a
- 802.15.4_mac_thrrfd_cm0_iar.a
- 802.15.4_mac_thrrfd_cm4_iar.a
- 802.15.4_mac_thr_cm0_iar.a
- 802.15.4_mac_thr_cm4_iar.a

WARNING

It is not recommended to use multiple MAC instances when the MAC is operating in Beacon mode.

6.17.1. Multiple PAN configuration

The functionality of the MPM layer is transparent to the upper layers (MAC, NWK, and so on). There is no need to call *MPM_Init()* explicitly, because the PHY layer initializes the Multiple PAN Manager at startup with the default configuration:

- Channel switch dwell time is 3.5 ms
- Automatic channel switch is disabled
- Active network is PAN0

The DualPanDwell consists of two components:

- Dwell Time Prescaler (bits [1:0])
- Actual Dwell Time, a multiple of the Prescaller time base (bits [7:2])

Table 18. Prescaler bits setting

Prescaller value bits [1:0]	Timebase	Range min	Range max
00	0.5 ms	0.5 ms	32 ms
01	2.5 ms	2.5 ms	160 ms
10	10 ms	10 ms	640 ms
11	50 ms	50 ms	3.2 s

The default value of the channel switch dwell timer is 3.5 ms:

```
#define mDefaultDualPanDwellPrescaller_c (0x00)
#define mDefaultDualPanDwellTime_c       (0x06)
```

If you want to change the MPM configuration at runtime, the *MPM_GetConfig()* and *MPM_GetConfig()* APIs can be used. If the PAN wants exclusive access to the PHY for a period of time, the *MPM_AcquirePAN()*, and *MPM_ReleasePAN()* APIs can be used. The acquired PAN is not interrupted by requests from other PANs. Until the PAN is released, other PANs cannot receive/transmit.
Each MAC instance can be configured to have a different role (Coordinator or EndDevice).

6.17.2. Multiple PAN SAPs

If the Multiple PAN feature is used, the application layer must either:

1. Implement multiple SAPs:
   - *MLME_NWK_PAN0_SapHandler()*, *MLME_NWK_PAN1_SapHandler()*
   - *MCPS_NWK_PAN0_SapHandler()*, *MCPS_NWK_PAN1_SapHandler()*
2. Discriminate the different PANs based on the `instanceId` parameter value.

The application is responsible for distinguishing between messages received on different PANs.
One approach is to use different queues for messages received in different SAPs (see the following example).

```
resultType_t MLME_NWK_SapHandler (nwkMessage_t* pMsg, instanceId_t instanceId)
{
    /* Put the incoming MLME message in the applications input queue. */
    MSG_Queue(&mMlmeNwkInputQueue[instanceId], pMsg);
    OSA_EventSet(&mAppEvent, gAppEvtMessageFromMLME_c);
    return gSuccess_c;
}
```

```
resultType_t MCPS_NWK_PAN1_SapHandler (mcpsToNwkMessage_t* pMsg, instanceId_t instanceId)
{
    /* Put the incoming MCPS message in the applications input queue. */
    MSG_Queue(&mMcpsNwkInputQueue[instanceId], pMsg);
    OSA_EventSet(&mAppEvent, gAppEvtMessageFromMCPS_c);
    return gSuccess_c;
}
```

6.17.3. Multiple PAN functionality

After reset, the hardware is configured to work in Multiple PAN Manual mode, with the default pan being PAN0. If the application doesn't perform any action on the second PAN, the device will work as if the second PAN does not exist. Before starting to operate in Multiple PAN mode, the application is responsible for setting the necessary parameters for all PANs:

- Extended Address (default is 0xFFFFFFFFFFFFFFFF)
- Channel Number (default is 0x0F)
- Mac Role (default is 0x00—EndDevice)
- PanId (default is 0xFFFF)
- Short Address (default is 0xFFFF)

NOTE

The above default values are identical for all PANs.

For example, the End Device operating in Multiple PAN mode can use the default values for PanId and ShortAddress until it associates with a Coordinator. When a device operates in Multiple PAN mode, the PANs can reside either on the same channel or on different channels. If the PANs are located on different channels, the device cannot be active on both networks at the same time. It must switch the Active Network either automatically or manually.

6.17.4. PANs located on the same channel

There is no need to switch the Active Network in this case. A received packet passes thorough both set of filters. If the packet satisfies at least one of them, then it is sent to the MAC.

A packet can satisfy all sets of addressing parameters (for example a broadcast packet). In this case, the MAC sends MCPS-DATA.Indication messages for every PAN.

6.17.5. PANs located on different channels

When the PANs are located on different channels and Rx is enabled by multiple MAC instances, the MPM periodically switches between the channels in search for a preamble. This automatic channel switching can be disabled if a PAN needs exclusive access to the PHY.

For example, if two MLME-SCAN.Request messages are queued from different networks at the same time, the MAC first executes the scan on PAN0. After the MLME-SCAN.Confirm is sent, the second scan operation is executed.

6.17.6. Multiple PAN feature scenarios

Because the MAC can be configured either as Coordinator or as EndDevice, there are three possible scenarios for Dual PAN operation:

- Both MAC instances are configured as EndDevices.
- One MAC instance is configured as Coordinator, and the other is configured as EndDevice.
- Both MAC instances are configured as Coordinators.

In the first scenario, the Rx/Tx operations of the two MAC instances never interfere because EndDevices periodically poll the Coordinator for data.

In the second scenario, Coordinators usually have the *RxOnWhenIdle* PIB set, which means that the Rx is able to receive data from other devices most of the time. When the MAC instance (configured as EndDevice) polls for data, the other MAC instance configured as Coordinator cannot receive.

In the last scenario, both MAC instances have the *RxOnWhenIdle* PIB set. When one MAC instance is transmitting data, the other MAC instance cannot receive.

When both instances must receive data, the MPM enables the automatic on-the-fly channel switch in search for a preamble. Once a preamble has been found, the channel switch is disabled unil the entire packet is received and acknowledged (if ACK was requested). During this period, the other MAC instance cannot receive.

6.17.7. Multiple PAN feature limitations

When operating in Multiple PAN Auto mode, the on-the-fly channel changes require approximately 68 μs. No preamble detection is possible during this “blind spot”.

The MAC always disables the automatic PAN switch during MLME-SCAN and MLME-POLL sequences. Other PANs cannot receive packets during this time even when the receiver is enabled, and transmissions are deferred until the first PAN finishes the sequence.

6.18. Coordinated Sampled Listening (CSL) feature

Coordinated Sampled Listening is a low-energy data-exchange mechanism that describes how the receiving devices periodically monitor channels to detect incoming transmissions. This feature is suitable for applications that require data transmission at an interval of approximately one second.

Coordinated Sampled Listening data transmission is performed by a PAN Coordinator when sending upper layer data to End Devices, while End Devices monitor the medium by performing channel samples (data reception).

This feature requires an overhead in each transmission by transmitting a series of Low-Energy Wakeup Frames that can be either long (for unsynchronized or broadcast transmissions), or short (if the MAC layer had previously obtained the listening period and timestamp of next channel sample for that destination). Coordinated Sampled Listening can be turned off to reduce transmission overhead by setting the CSL PIBs to 0.

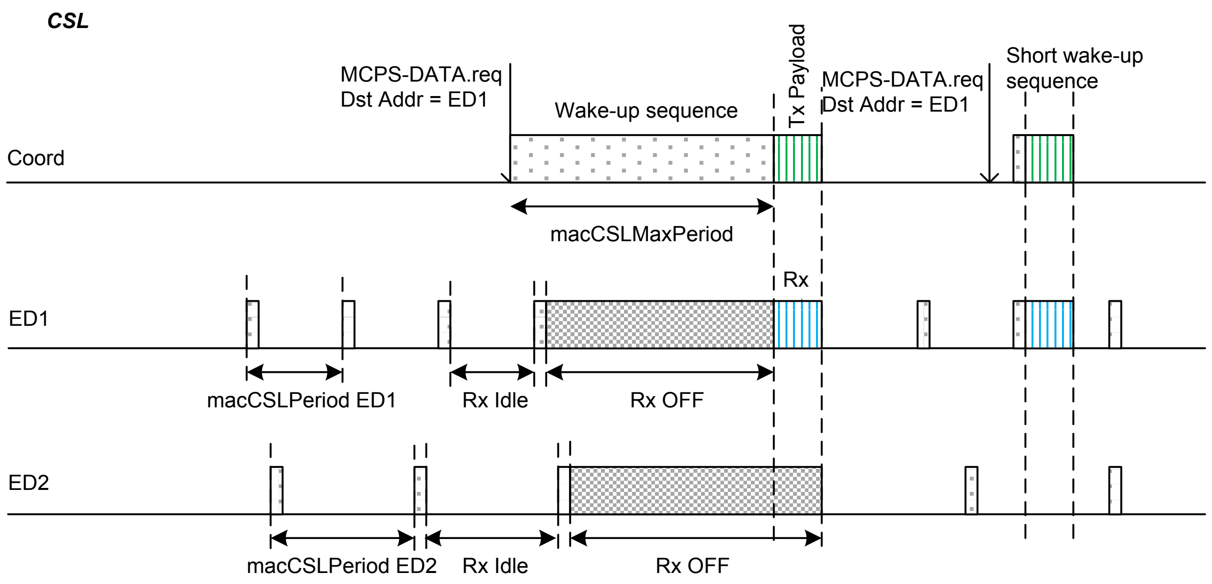

Figure 6. Coordinated Sampled Listening

6.18.1. CSL Reception

CSL Reception is enabled on an End Device by issuing MLME-SET.Request on the macCSLPeriod PIB. The value is measured in units of 10 symbols, and specifies how often the device performs a channel sample. The channel sample is represented by Energy Detection scan of CCA Duration symbols.

If the reported energy is above a predefined threshold, the receiver is enabled to receive Low-Energy Wakeup Frame. Otherwise, the receiver is turned off until the next channel sample is performed.

When Low-Energy Wakeup Frame is received and the destination address matches receiving device’s address, the receiver is turned off for the rendezvous time specified in the packet (this is the time difference between the wakeup frame timestamp and the upper layer data frame timestamp).
When the Coordinator application data frame is received, the End Device responds with enhanced acknowledgement frame, where it includes the Low-Energy CSL Information Element.

If the application data frame contains Frame Pending flag set, the receiver is enabled immediately after sending the Enhanced Acknowledgment to receive another application data frame.

If the destination information in the Wakeup Frame do not match the receiving device’s address, the receiver is turned off for the rendezvous timestamp plus the transmission duration of a maximum sized packet plus enhanced acknowledgement duration, after which channel sample is performed again.

6.18.2. CSL Rx configuration example

Example 14. CSL Rx configuration

```
/* value expressed in units of 10 MAC symbols - 1 second */
#define gMPibCslPeriodValue_c    0x1388 // 1 sec

void App_ConfigureMAC_LE_PIBs()
{
    uint32_t value;

    /* Message for the MLME will be allocated and attached to this pointer */
    mlmeMessage_t *pMsg;

    /* Allocate a message for the MLME (We should check for NULL). */
    pMsg = MSG_AllocType(mlmeMessage_t);
    if( pMsg != NULL )
    {
        /* Configure addressing PIBs */
        /* MAC short address */
        pMsg->msgType = gMlmeSetReq_c;
        pMsg->msgData.setReq.pibAttribute = gMPibShortAddress_c;
        pMsg->msgData.setReq.pibAttributeValue = (uint16_t*)&macShortAddress;
        (void)NWK_MLME_SapHandler( pMsg, mMacInstance );

        /* MAC PAN ID */
        pMsg->msgType = gMlmeSetReq_c;
        pMsg->msgData.setReq.pibAttribute = gMPibPanId_c;
        pMsg->msgData.setReq.pibAttributeValue = (uint16_t*)&macPanId;
        (void)NWK_MLME_SapHandler( pMsg, mMacInstance );

        /* MAC logical channel */
        pMsg->msgType = gMlmeSetReq_c;
        pMsg->msgData.setReq.pibAttribute = gMPibLogicalChannel_c;
        pMsg->msgData.setReq.pibAttributeValue = (uint8_t*)&macLogicalChannel;
        (void)NWK_MLME_SapHandler( pMsg, mMacInstance );

        /* Initialize the MAC CSL Period */
        pMsg->msgType = gMlmeSetReq_c;
        pMsg->msgData.setReq.pibAttribute = gMPibCslPeriod_c;
        value = gMPibCslPeriodValue_c;
        pMsg->msgData.setReq.pibAttributeValue = (uint16_t*)&value;
        (void)NWK_MLME_SapHandler( pMsg, mMacInstance );

        /* Free message, all requests were sync */
        MSG_Free(pMsg);
    }
}
```

6.18.3. CSL Transmission

CSL Transmission is enabled on a PAN Coordinator by issuing MLME-SET.Request on the macCSLMaxPeriod PIB. The value is set in units of 10 symbols and must be set larger than any other CSL period set on any of the End Devices. The specified value is the duration of a long wakeup sequence which overlaps all channel samples, so that a broadcast packet or an unsynchronized transmission is received by the specified destination.

NOTE

macCslMaxPeriod must be set at a value greater than any other macCslPeriod set on any of the network participants.

When enabled, CSL transmission generates a long wakeup sequence at the first MCPS-DATA.Request issued for a destination. The wakeup sequence is a series of wakeup frames transmitted by the PAN Coordinator that announce the moment of transmission of upper layer data frame to a destination. This sequence starts with CSMA-CA for any but the first frame of the sequence. After the upper layer data frame is transmitted, the receiver is enabled for macEnhAckWaitDuration.

Synchronization is made by adding the destination CSL phase to the timestamp of the enhanced acknowledgment, and storing the next destination channel sample. This information (along with the CSL period of the device) is computed for future transactions to the same destination.

When receiving MCPS-DATA.Request for which the MAC layer had previously stored channel sampling information, the packet is stored into the CSL Tx queue, and it is scheduled for transmission at the computed destination channel sample time. This is called synchronized transmission and requires a short wakeup duration that is long enough to overlap the destination channel sample and announce rendezvous time through a wakeup frame.

When a packet is inserted into the CSL queue for a synchronized transmission, MAC layer searches for packets already inserted in queue for the same destination, and updates the frame pending flag for that packet. After each enhanced acknowledgement is received, the CSL queue is searched for packets for the same destination. This way the throughput is increased, and no future wakeup sequences are required to be scheduled.

CSL retransmissions are performed for up to macMaxFrameRetries in two ways: immediately for long retransmissions for unsynchronized/broadcast transmissions, or scheduled short wakeup sequences for synchronized transmissions.

If a synchronized transmission fails to receive an enhanced acknowledgement after macMaxFrameRetries, the synchronization entry in the CSL table is marked as not usable and the next MCPS-DATA.Request for the same destination resorts to a long wakeup sequence.

6.18.4. CSL Tx configuration example

Example 15. CSL Tx configuration

```
/* value expressed in units of 10 MAC symbols - 1.5 seconds */
#define gMPibCslMaxPeriodValue_c 0x1D4C

void App_ConfigureMAC_LE_PIBs()
{
    uint32_t value;

    /* Message for the MLME will be allocated and attached to this pointer */
    mlmeMessage_t *pMsg;

    /* Allocate a message for the MLME (We should check for NULL). */
    pMsg = MSG_AllocType(mlmeMessage_t);
    if( pMsg != NULL )
    {
        /* Initialize the MAC CSL Max Period */
        pMsg->msgType = gMlmeSetReq_c;
        pMsg->msgData.setReq.pibAttribute = gMPibCslMaxPeriod_c;
        value = gMPibCslMaxPeriodValue_c;
        pMsg->msgData.setReq.pibAttributeValue = (uint16_t*)&value;
        (void)NWK_MLME_SapHandler( pMsg, mMacInstance );

        /* Free message, all requests were sync */
        MSG_Free(pMsg);
    }
}
```

6.18.5. CSL limitations

- PAN Coordinator does not enable CSL reception; it enables its receiver upon the value of macRxOnWhenIdle PIB.
- CSL is enabled after the PAN Coordinator starts the network and all network devices that require association to the network are associated.
- CSL transmission requires issuing MCPS-DATA.Request with the ACK required flag set, except for when the destination addressing is broadcast.
- CSL transmission requires MCPS-DATA.Request to use short addressing mode, as this is the only mode supported by the Low-Energy Wakeup Frame.

6.19. Receiver-Initiated Transmission (RIT) feature

Receiver-Initiated Transmission is a low-energy feature that describes how the receiving devices periodically send RIT Data Request Commands to announce their receiver-enabled periods to other devices. Receiver-Initiated Transmission is suitable for applications that require data exchange rate of tens of seconds due to the amount of energy required by the RIT Data Request transmission.

Receiver-Initiated Transmission has two types of listening modes, depending on whether the RIT Information Element is set or not (with or without listening schedule). RIT is enabled by issuing a MLME-SET.Request on the macRitPeriod PIB, whose value is measured in units of aBaseSuperframeDuration and the MAC layer immediately starts transmitting RIT Data Request at this interval.

NOTE

The macRitDataWaitDuration, macRitTxWaitDuration (and macRitIe optionally) must be set before enabling RIT through macRitPeriod.

6.19.1. RIT reception without listening schedule

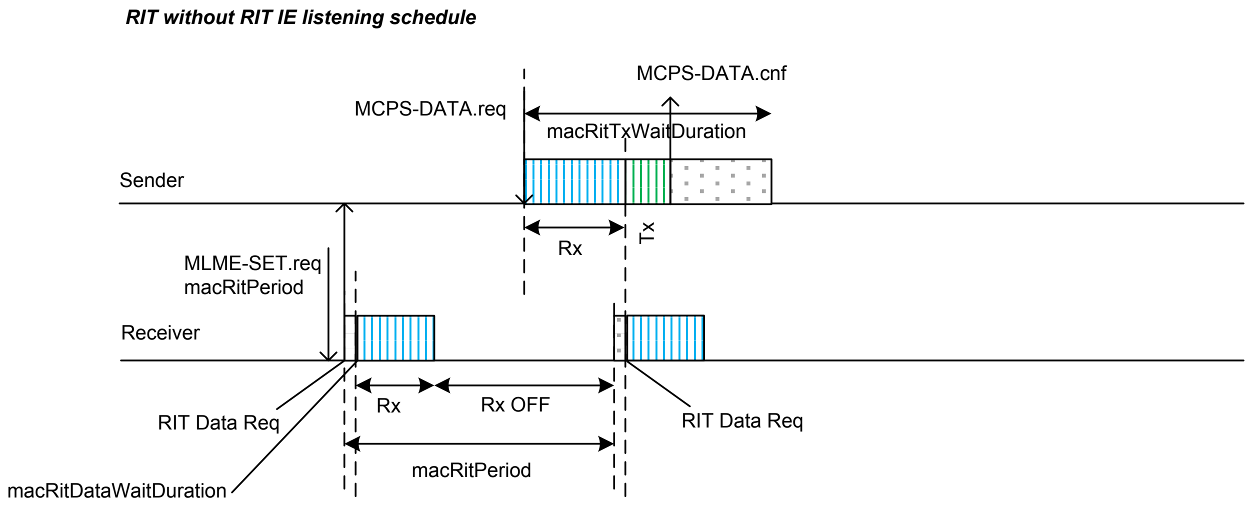

Figure 7. RIT without listening schedule

When RIT is enabled and macRitIe is not set, there is no listening schedule available. The MAC layer starts transmitting RIT Data Request command frames with their associated PAN coordinator addresses on the End Device, or broadcast address on a PAN Coordinator using CSMA-CA. After each RIT Data Request frame, the receiver is enabled for macRitDataWaitDuration, which is also measured in units of aBaseSuperframeDuration. It then turns the receiver off until the next RIT Data Request command is to be sent.

6.19.2. RIT configuration example without RIT IE

Example 16. RIT configuration without listening schedule

```
#define gMPibRitTxWaitDurationValue_c    0x4E2 // 24 sec
#define gMPibRitDataWaitDurationValue_c  0x34  //  1 sec
#define gMPibRitPeriodValue_c            0x410 // 20 sec

void App_ConfigureMAC_LE_PIBs()
{
    uint32_t value;

    /* Message for the MLME will be allocated and attached to this pointer */
    mlmeMessage_t *pMsg;

    /* Allocate a message for the MLME */
    pMsg = MEM_BufferAlloc(mlmeMessage_t);
    if( pMsg != NULL )
    {
```

```
/* Configure addressing PIBs */
/* MAC short address */
pMsg->msgType = gMlmeSetReq_c;
pMsg->msgData.setReq.pibAttribute = gMPibShortAddress_c;
pMsg->msgData.setReq.pibAttributeValue = (uint16_t*)&macShortAddress;
(void)NWK_MLME_SapHandler( pMsg, mMacInstance );

/* MAC PAN ID */
pMsg->msgType = gMlmeSetReq_c;
pMsg->msgData.setReq.pibAttribute = gMPibPanId_c;
pMsg->msgData.setReq.pibAttributeValue = (uint16_t*)&macPanId;
(void)NWK_MLME_SapHandler( pMsg, mMacInstance );

/* MAC logical channel */
pMsg->msgType = gMlmeSetReq_c;
pMsg->msgData.setReq.pibAttribute = gMPibLogicalChannel_c;
pMsg->msgData.setReq.pibAttributeValue = (uint8_t*)&macLogicalChannel;
(void)NWK_MLME_SapHandler( pMsg, mMacInstance );

/* Initialize the MAC RIT Tx Wait Duration */
pMsg->msgType = gMlmeSetReq_c;
pMsg->msgData.setReq.pibAttribute = gMPibRitTxWaitDuration_c;
value = gMPibRitTxWaitDurationValue_c;
pMsg->msgData.setReq.pibAttributeValue = (uint32_t*)&value;
(void)NWK_MLME_SapHandler( pMsg, mMacInstance );

/* Initialize the MAC RIT Data Wait Duration */
pMsg->msgType = gMlmeSetReq_c;
pMsg->msgData.setReq.pibAttribute = gMPibRitDataWaitDuration_c;
value = gMPibRitDataWaitDurationValue_c;
pMsg->msgData.setReq.pibAttributeValue = (uint8_t*)&value;
(void)NWK_MLME_SapHandler( pMsg, mMacInstance );

/* Initialize the MAC RIT Period */
pMsg->msgType = gMlmeSetReq_c;
pMsg->msgData.setReq.pibAttribute = gMPibRitPeriod_c;
value = gMPibRitPeriodValue_c;
pMsg->msgData.setReq.pibAttributeValue = (uint32_t*)&value;
(void)NWK_MLME_SapHandler( pMsg, mMacInstance );
```

```
        /* Free message, all requests were sync */
        MSG_Free(pMsg);
    }
}
```

6.19.3. RIT reception with listening schedule

RIT with RIT IE listening schedule

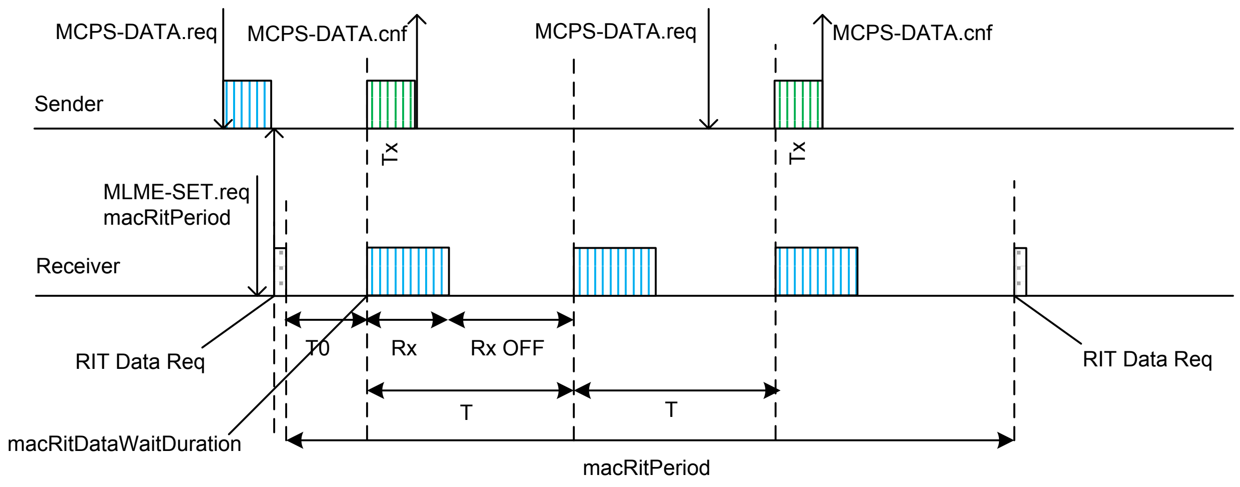

Figure 8. RIT with listening schedule

The RIT Data Request Command can include a four-octet payload, represented by the macRitIe PIB value. If the RIT information element is set, the receiver is enabled based on the listening schedule, as described in these steps:

1. RIT Data Request command is sent
2. Receiver is disabled for T0 symbols
3. Receiver is enabled every T symbols for macRitDataWaitDuration symbols
4. Step 3 is repeated N times
5. Repeat from Step 1

NOTE

RIT Information Elements T0 and T are measured in the same unit as macRitPeriod (aBaseSuperframeDuration).

6.19.4. RIT configuration example with RIT IE

Example 17. RIT configuration with listening schedule

```
#define gMPibRitTxWaitDurationValue_c    0x4E2 // 24 sec
#define gMPibRitDataWaitDurationValue_c  0x34  //  1 sec
#define gMPibRitPeriodValue_c            0x410 // 20 sec
#define gMPibRitIeT0_c                   0x9C  //  3 sec
#define gMPibRitIeT_c                    0x104 //  5 sec
#define gMPibRitIeN_c                    0x03  //  3 Rx On repeats

void App_ConfigureMAC_LE_PIBs()
{
    uint32_t value;

    /* Message for the MLME will be allocated and attached to this pointer */
    mlmeMessage_t *pMsg;

    /* Allocate a message for the MLME */
    pMsg = MEM_BufferAlloc(mlmeMessage_t);
    if( pMsg != NULL )
    {
        /* Configure addressing PIBs */
        /* MAC short address */
        pMsg->msgType = gMlmeSetReq_c;
        pMsg->msgData.setReq.pibAttribute = gMPibShortAddress_c;
        pMsg->msgData.setReq.pibAttributeValue = (uint16_t*)&macShortAddress;
        (void)NWK_MLME_SapHandler( pMsg, mMacInstance );

        /* MAC PAN ID */
        pMsg->msgType = gMlmeSetReq_c;
        pMsg->msgData.setReq.pibAttribute = gMPibPanId_c;
        pMsg->msgData.setReq.pibAttributeValue = (uint16_t*)&macPanId;
        (void)NWK_MLME_SapHandler( pMsg, mMacInstance );

        /* MAC logical channel */
        pMsg->msgType = gMlmeSetReq_c;
        pMsg->msgData.setReq.pibAttribute = gMPibLogicalChannel_c;
        pMsg->msgData.setReq.pibAttributeValue = (uint8_t*)&macLogicalChannel;
        (void)NWK_MLME_SapHandler( pMsg, mMacInstance );
        /* Initialize the MAC RIT Tx Wait Duration */
        pMsg->msgType = gMlmeSetReq_c;
        pMsg->msgData.setReq.pibAttribute = gMPibRitTxWaitDuration_c;
        value = gMPibRitTxWaitDurationValue_c;
        pMsg->msgData.setReq.pibAttributeValue = (uint32_t*)&value;
        (void)NWK_MLME_SapHandler( pMsg, mMacInstance );

        /* Initialize the MAC RIT Data Wait Duration */
        pMsg->msgType = gMlmeSetReq_c;
        pMsg->msgData.setReq.pibAttribute = gMPibRitDataWaitDuration_c;
        value = gMPibRitDataWaitDurationValue_c;
        pMsg->msgData.setReq.pibAttributeValue = (uint8_t*)&value;
        (void)NWK_MLME_SapHandler( pMsg, mMacInstance );

        /* Initialize the MAC RIT IE Listening schedule */
        pMsg->msgType = gMlmeSetReq_c;
        pMsg->msgData.setReq.pibAttribute = gMPibRitIe_c;
        ((macRitIe_t*)&value)->T0 = gMPibRitIeT0_c;
        ((macRitIe_t*)&value)->N = gMPibRitIeN_c;
        ((macRitIe_t*)&value)->T = gMPibRitIeT_c;
        pMsg->msgData.setReq.pibAttributeValue = (macRitIe_t*)&value;
        (void)NWK_MLME_SapHandler( pMsg, mMacInstance );

        /* Initialize the MAC RIT Period */
        pMsg->msgType = gMlmeSetReq_c;
        pMsg->msgData.setReq.pibAttribute = gMPibRitPeriod_c;
        value = gMPibRitPeriodValue_c;
        pMsg->msgData.setReq.pibAttributeValue = (uint32_t*)&value;
        (void)NWK_MLME_SapHandler( pMsg, mMacInstance );

        /* Free message, all requests were sync */
        MSG_Free(pMsg);
    }
}
```

6.19.5. RIT transmission

A device that receives MCPS-DATA.Request first stops its periodic RIT Data Request commands. Then, it enables the receiver for macRitTxWaitDuration, waiting for RIT Data Request from the destination device. During this interval, any frame other than RIT Data Request is ignored.

NOTE

macRitTxWaitDuration must be set at a value greater than any other macRitPeriod set on any of the network participants.

If the received RIT Data Request frame carries RIT Information Element with listening schedule as payload, this information and the RIT Data Request timestamp are stored in the RIT table for future use.

If the received RIT Data Request frame matches the MCPS-DATA.Request destination addressing information, the MAC layer immediately sends the upper-layer data (if there is no listening schedule provided), or inserts the packet into the RIT Tx queue and schedules a transmission to the nearest Rx On period (as per destination listening schedule).

If MCPS-DATA.Request is received, the MAC layer first checks if there is a listening schedule for the destination. If found, it checks if there is an Rx On interval available, depending on the number of repeat listen stored (N). If these conditions are met, the packet is stored in the RIT Tx queue and is scheduled for transmission at the nearest available Rx On interval.

If the above conditions are not met, the destination is marked as no longer available for synchronized transmissions, and the MAC layer resorts to enabling its receiver for macRitTxWaitDuration period to receive another RIT Data Request command from the destination device.

All RIT retransmissions are performed as soon as possible. If macMaxFrameRetries number of retries is reached and the transmission was synchronized to an Rx On interval, the destination schedule is marked as no longer available for use.

6.19.6. RIT limitations

Receiver-Initiated Transmission is enabled on a network only after the PAN Coordinator starts the network, and the devices for which the communication is intended are associated to the network.

All MCPS-DATA.Requests issued when the RIT is enabled must use short addressing mode, as the RIT Data Requests use the same addressing mode.

Broadcast transmissions are not available when the RIT is enabled.

6.20. Time-Slotted Channel Hopping (TSCH) feature

Time-Slotted Channel Hopping enables devices to perform frequency hopping in time-division manner, specified by a timeslot. The duration of a timeslot enables a single exchange of a MAC frame, and eventually an acknowledgment between two nodes. Multiple frames cannot be exchanged in the same timeslot.

6.20.1. TSCH configuration elements

6.20.1.1. Slotframes

Slotframe is a collection of timeslots reapeating in time. Slotframe is represented by a slotframe handle and its size. The size of a slotframe specifies when each timeslot of the frame is repeated again. Multiple slotframes with various sizes can be set on a device. This enables various communication patterns to be configured. Lower slotframe handles have higher priority than higher slotframe handles, while any transmission has higher priority than reception, regardless of the slotframe handle.

The coordinator that starts the network and enables TSCH sets the absolute slot number inside the network, which is used in synchronization to the network by other joining devices. The absolute slot number represents the number of timeslots that have elapsed after starting the network. It is also used as a part of the CCM* nonce when security and TSCH are enabled, instead of the usual frame counter. This introduces a time-dependent component into the security process, as the ASN increments with every timeslot that passes, rather than incrementing only when sending frames. Slotframes are set using the MLME-SET-SLOTFRAME.Request primitive.

6.20.1.2. Links

Link is an oriented communication between two nodes, taking place on a certain timeslot with a certain channel offset. Links are a pairwise communication between nodes, with one of the peers assigned to transmit and the other to be in reception.

Links are assigned to slotframes, and every link handle must be provided with a slotframe handle. Other parameters of a link are a timeslot and a channel offset. The timeslot assigned is the timeslot inside the slotframe when the link is available for use, and the channel offset is used to computate the channel on which the communication takes place. Links can be of multiple types:

- Tx Links—used for performing transmission
- Rx Links—used for performing reception
- Normal Links—used for sending MAC data frames
- Advertising Links—used for sending MAC Enhanced Beacons and/or MAC data frames
- Shared Links(Tx)—allocated to more than one device for Tx
- Timekeeping Links(Rx)—used for maintaining clock source from a peer node

Certain combinations have no meaning (for example Shared Rx Links or Timekeeping Tx Links). Links are set using the MLME-SET-LINK.Request primitive.

6.20.1.3. TSCH channel hopping

Channels can be configured by setting two PIBs:

- gMPibHoppingSequenceLength_c—length of the channel-hopping sequence
- gMPibHoppingSequenceList_c—list of channels in the hopping sequence

Channel hopping can include the same channel multiple times. The current channel in a timeslot is calculated using the slotframe and link set for that timeslot, following this formula:

Channel = HoppingSequenceList [(ASN + link channel offset) % HoppingSequenceLength]

Note that a channel is set when (and only when) a link from any slotframe is selected for that timeslot to perform either transmission or reception. The channel offset is present to give each timeslot a number of up to HoppingSequenceLength synchronous communications on different channels, so that the node communication matrix inside a slotframe is slotframe size in timeslots multiplied by the hopping sequence length channels.

6.20.1.4. Network joining

Timeslot communication makes the MAC Active Scan procedure to have almost zero chances to succeed. When TSCH is enabled, the recommended way to join devices to network is to perform Passive Scan. This requires the higher layer of the coordinator to set an advertising procedure.
This represents the set procedure of a Tx link, with the advertising type set, and a broadcast address as the node address. When the higher layer of Coordinator decides to advertise the parameters required for a node to join the network, it must issue the MLME-BEACON.Request primitive, with Enhanced Beacon type, and a channel from the channel-hopping sequence to send the beacon on. Note that the address from the MLME-BEACON.Request primitive must match the node address from the advertising Tx link set, and short addressing must be used due to the node address short addressing mode.

The TSCH parameters included in the Enhanced Beacon as Information Elements by default are:

- gMacPayloadIeIdChannelHoppingSequence_c
- gMacPayloadIeIdTschSynchronization_c
- gMacPayloadIeIdTschSlotframeAndLink_c
- gMacPayloadIeIdTschTimeslot_c

These are set as default parameters of the MAC EB IE List PIB when TSCH is enabled. They can be configured using MLME-SET.Request primitive afterwards (if higher layer decides so), but most of them are required to join the process for a new node to be synchronized. It is not recommended to modify the TSCH Timeslot PIB, as the default timings provided are the recommended ones.

The End Device is performing Passive Scan on a channel mask that must include the advertising channel used by the PAN Coordinator, and also be a part of the channel-hopping sequence of the network.
The timeslot start synchronization is performed by the MAC layer, based on the Enhanced beacon timestamp and the Coordinator timeslot timings (when TSCH is enabled). The higher layer is presented the IEs and the beacon payload, and it is responsible for configuring the slotframe and link parameters to be used inside the network.

The number of coordinators found during a scan can be larger than one. The MAC layer is responsible for storing their information in case the higher layer decides to choose between them. The allocation of memory is performed dynamically, but it is limited to the number configured using gMacTschMaxPanCoordSync_c. These buffers are freed once TSCH is enabled and synchronization is performed, as it is considered that PAN Coordinators are no longer needed.

6.20.1.5. Clock synchronization

It is highly recommended to use at least one timekeeping Rx link for every new device that joins the network. A clock source for each node is required to correct any clock drifts that may appear between devices, and properly detect the timeslot start from the PAN Coordinator perspective.

Timeslot start is very important for the precision of the actions performed by both devices exchanging a frame in the timeslot. Time perspective is exchanged using MAC data frames or Enhanced Acknowledgments through the ACK/NACK Information element. The expected time to receive a MAC frame or an ACK frame is compared with the timestamp of the frame and adjusted locally, or communicated through an EACK frame to the other node.

A node can have more than one peer as a time source. When multiple nodes are configured as such, the node adjusts its timeslot start with a fraction of one, divided by the number of clock sources for each adjustment made after frame exchange.

If no frames are exchanged by the higher layer of two nodes (one being a clock source for the other), the node with the timekeeping Rx link sends MAC Keepalive frame with an ACK required flag set, requiring the clock source to specify its time perspective. Keeping the period after which this exchange is performed alive is configured by the higher layer using MLME-KEEP-ALIVE.request primitive, and it is expressed in timeslots. Note that this primitive must be issued after the Rx Link with Keepalive option is set on the node.

6.20.1.6. TSCH enablement

Before enabling TSCH, the TSCH Role must be set on the device. The TSCH PAN is started (ASN = 0) by a TSCH PAN Coordinator (usually the PAN Coordinator). The other nodes join the TSCH network that is already started, and they must be configured as TSCH devices through the TSCH role. The TSCH can be enabled using TSCH-MODE.Request primitive with TSCH mode parameter set to On, and disabled through Off.

6.20.2. TSCH primitives

6.20.2.1. Set-slotframe-request

The Set-slotframe-request structure is:

```
// Type: gMlmeSetSlotframeReq_c,
typedef MAC_STRUCT mlmeSetSlotframeReq_tag
{
    uint8_t                 slotframeHandle;
    macSetSlotframeOp_t     operation;
    uint16_t                size;
} mlmeSetSlotframeReq_t;
```

6.20.2.2. Set-slotframe-confirm

The Set-slotframe-confirm structure is:

```
// Type: gMlmeSetSlotframeCnf_c,
typedef MAC_STRUCT mlmeSetSlotframeCnf_tag
{
    uint8_t                 slotframeHandle;
    resultType_t            status;
} mlmeSetSlotframeCnf_t;
```

6.20.2.3. Set-link-request

The Set-link-request structure is:

```
// Type: gMlmeSetLinkReq_c,
typedef MAC_STRUCT mlmeSetLinkReq_tag
{
    macSetLinkOp_t          operation;
    uint16_t                linkHandle;
    uint8_t                 slotframeHandle;
    uint16_t                timeslot;
    uint16_t                channelOffset;
    macLinkOptions_t        linkOptions;
    macLinkType_t           linkType;
    uint16_t                nodeAddr;
} mlmeSetLinkReq_t;
```

6.20.2.4. Set-link-confirm

The Set-link-confirm structure is:

```
// Type: gMlmeSetLinkCnf_c,
typedef MAC_STRUCT mlmeSetLinkCnf_tag
{
    resultType_t            status;
    uint16_t                linkHandle;
    uint8_t                 slotframeHandle;
} mlmeSetLinkCnf_t;
```

6.20.2.5. Tsch-mode-request

The Tsch-mode-request structure is:

```
// Type: gMlmeTschModeReq_c,
typedef MAC_STRUCT mlmeTschModeReq_tag
{
    macTschMode_t           tschMode;
} mlmeTschModeReq_t;
```

6.20.2.6. Tsch-mode-confirm

The Tsch-mode-confirm structure is:

```
// Type: gMlmeTschModeCnf_c,
typedef MAC_STRUCT mlmeTschModeCnf_tag
{
    macTschMode_t           tschMode;
    resultType_t            status;
} mlmeTschModeCnf_t;
```

6.20.2.7. Keep-alive-request

The Keep-alive-request structure is:

```
// Type: gMlmeKeepAliveReq_c,
typedef MAC_STRUCT mlmeKeepAliveReq_tag
{
    uint16_t                dstAddr;
    uint16_t                keepAlivePeriod;
} mlmeKeepAliveReq_t;
```

6.20.2.8. Keep-alive-confirm

The Keep-alive-confirm structure is:

```
// Type: gMlmeKeepAliveCnf_c,
typedef MAC_STRUCT mlmeKeepAliveCnf_tag
{
    resultType_t            status;
} mlmeKeepAliveCnf_t;
```

6.20.2.9. Beacon-request

The Beacon-request structure is:

```
// Type: gMlmeBeaconReq_c,
typedef MAC_STRUCT mlmeBeaconReq_tag
{
    beaconType_t            beaconType;
    uint8_t                 channel;
    channelPageId_t         channelPage;
    uint8_t                 superframeOrder;
    macSecurityLevel_t      beaconSecurityLevel;
    keyIdModeType_t         beaconKeyIdMode;
    uint64_t                beaconKeySource;
    uint8_t                 beaconKeyIndex;
    addrModeType_t          dstAddrMode;
    uint64_t                dstAddr;
    bool_t                  bsnSuppression;
} mlmeBeaconReq_t;
```

6.20.2.10. Beacon-confirm

The Beacon-confirm structure is:

```
// Type: gMlmeBeaconCnf_c,
typedef MAC_STRUCT mlmeBeaconCnf_tag
{
    resultType_t            status;
} mlmeBeaconCnf_t;
```

6.20.3. TSCH configuration on a coordinator

The TSCH configuration parameters header file is shown in this example:

Example 18. TSCH parameters for coordinator

```
/* Hopping sequence */
#define mMacHoppingSequenceId_c         0
#define mMacHoppingSequenceLen_c        3
#define mMacHoppingSequenceList_c       { 0, 1, 2 }

/* Slotframes */
const macSlotframeIe_t slotframeArray[] =
{
  {
    .macSlotframeHandle = 0,
    .macSlotframeSize = 10,
  },
};

/* Links */
const macLink_t linkArray[] =
{
    {
        .macLinkHandle = 0,
        .macNodeAddress = 0xFFFF,
        .macLinkType = gMacLinkTypeAdvertising_c,
        .slotframeHandle = 0,
        .macLinkIe = {
                        .timeslot = 0,
                        .channelOffset = 0,
                        .macLinkOptions =
                        {
                          .tx = 1,
                        },
                     },
    },
    {
        .macLinkHandle = 1,
        .macNodeAddress = mDefaultValueOfEDShortAddr_c,
        .macLinkType = gMacLinkTypeNormal_c,
        .slotframeHandle = 0,
        .macLinkIe = {
                        .timeslot = 1,
                        .channelOffset = 0,
                        .macLinkOptions =
                        {
                          .tx = 1,
                        },
                    },
  },
  {
        .macLinkHandle = 2,
        .macNodeAddress = mDefaultValueOfEDShortAddr_c,
        .macLinkType = gMacLinkTypeNormal_c,
        .slotframeHandle = 0,
        .macLinkIe = {
                        .timeslot = 2,
                        .channelOffset = 0,
                        .macLinkOptions =
                        {
                          .rx = 1,
                        },
                    },
  },
};
```

The TSCH primitives configuration is shown in this example:

Example 19. TSCH configuration on coordinator

```
static uint8_t App_Configure_MAC_TSCH_Params(void)
{
    uint32_t i;
    uint16_t hoppingSequenceLength = mMacHoppingSequenceLen_c;
    uint8_t mHoppingSequenceList[] = mMacHoppingSequenceList_c;
    macTschRole_t tschRole = gMacTschRolePANCoordinator_c;

    /* Message for the MLME will be allocated and attached to this pointer */
    mlmeMessage_t *pMsg = MSG_AllocType(mlmeMessage_t);

    if( pMsg != NULL )
    {
        /* Set MAC Hopping Sequence Length */
        pMsg->msgType = gMlmeSetReq_c;
        pMsg->msgData.setReq.pibAttribute = gMPibHoppingSequenceLength_c;
        pMsg->msgData.setReq.pibAttributeValue = &hoppingSequenceLength;
        (void)NWK_MLME_SapHandler( pMsg, mMacInstance );

        /* Set MAC Hopping Sequence List */
        pMsg->msgType = gMlmeSetReq_c;
        pMsg->msgData.setReq.pibAttribute = gMPibHoppingSequenceList_c;
        pMsg->msgData.setReq.pibAttributeValue = mHoppingSequenceList;
        (void)NWK_MLME_SapHandler( pMsg, mMacInstance );

        /* Set slotframes */
        for( i=0; i<sizeof(slotframeArray)/sizeof(macSlotframeIe_t); i++ )
        {
            pMsg->msgType = gMlmeSetSlotframeReq_c;
            pMsg->msgData.setSlotframeReq.operation = gMacSetSlotframeOpAdd_c;
            pMsg->msgData.setSlotframeReq.slotframeHandle =
slotframeArray[i].macSlotframeHandle;
            pMsg->msgData.setSlotframeReq.size = slotframeArray[i].macSlotframeSize;
            (void)NWK_MLME_SapHandler( pMsg, mMacInstance );
        }

        /* Set links */
        for( i=0; i<sizeof(linkArray)/sizeof(macLink_t); i++ )
        {
            pMsg->msgType = gMlmeSetLinkReq_c;
            pMsg->msgData.setLinkReq.operation = gMacSetLinkOpAdd_c;
            pMsg->msgData.setLinkReq.slotframeHandle = linkArray[i].slotframeHandle;
            pMsg->msgData.setLinkReq.linkHandle = linkArray[i].macLinkHandle;
            pMsg->msgData.setLinkReq.linkType = linkArray[i].macLinkType;
            pMsg->msgData.setLinkReq.nodeAddr = linkArray[i].macNodeAddress;
            pMsg->msgData.setLinkReq.timeslot = linkArray[i].macLinkIe.timeslot;
            pMsg->msgData.setLinkReq.channelOffset = linkArray[i].macLinkIe.channelOffset;
            pMsg->msgData.setLinkReq.linkOptions = linkArray[i].macLinkIe.macLinkOptions;
            (void)NWK_MLME_SapHandler( pMsg, mMacInstance );
        }

       /* Set TSCH Role as PAN Coordinator */
```

```
        pMsg->msgType = gMlmeSetReq_c;
        pMsg->msgData.setReq.pibAttribute = gMPibTschRole_c;
        pMsg->msgData.setReq.pibAttributeValue = &tschRole;
        (void)NWK_MLME_SapHandler( pMsg, mMacInstance );
        /* Enable TSCH */
        pMsg->msgType = gMlmeTschModeReq_c;
        pMsg->msgData.tschModeReq.tschMode = gMacTschModeOn_c;
        (void)NWK_MLME_SapHandler( pMsg, mMacInstance );

        /* Free message, all requests were sync */
        MSG_Free(pMsg);
    }
    else
    {
        /* Allocation of a message buffer failed. */
        return errorAllocFailed;
    }

    return errorNoError;
}
```

The Coordinator is recommended to periodically advertise network parameters to enable new devices to join the network. This is done through the MLME-BEACON.Request primitive, as shown in this example:

Example 20. TSCH advertising on coordinator

```
    /* Message for the MLME will be allocated and attached to this pointer */
    mlmeMessage_t *pMsg = MSG_Alloc(sizeof(mlmeMessage_t) + gMaxPHYPacketSize_c);
    uint8_t mHoppingSequenceList[] = mMacHoppingSequenceList_c;

    if(pMsg != NULL)
    {
        /* Create MLME-BEACON Request message. */
        pMsg->msgType = gMlmeBeaconReq_c;
        pMsg->msgData.beaconReq.beaconType = gMacEnhancedBeacon_c;
        pMsg->msgData.beaconReq.channel = mHoppingSequenceList[0];
        pMsg->msgData.beaconReq.channelPage = gChannelPageId9_c;
        pMsg->msgData.beaconReq.superframeOrder = 0x0F;
        pMsg->msgData.beaconReq.beaconSecurityLevel = gMacSecurityNone_c;
        pMsg->msgData.beaconReq.dstAddrMode = gAddrModeShortAddress_c;
        pMsg->msgData.beaconReq.dstAddr = 0xFFFF;
        pMsg->msgData.beaconReq.bsnSuppression = FALSE;

        (void)NWK_MLME_SapHandler( pMsg, mMacInstance );

        mcPendingPackets++;
    }
```

6.20.4. TSCH configuration on an end-device

The TSCH configuration parameters header file is shown in this example:

Example 21. TSCH parameters for an end-device

```
/* Hopping sequence */
#define mMacHoppingSequenceId_c         0
#define mMacHoppingSequenceLen_c        3
#define mMacHoppingSequenceList_c       { 0, 1, 2 }

#define mMacKeepAlivePeriod_c           1000    /* timeslots */

/* Slotframes */
const macSlotframeIe_t slotframeArray[] =
{
  {
    .macSlotframeHandle = 0,
    .macSlotframeSize = 10,
  },
};

/* Links */
const macLink_t linkArray[] =
{
    {
        .macLinkHandle = 0,
        .macNodeAddress = mDefaultValueOfCoordAddress_c,
        .macLinkType = gMacLinkTypeNormal_c,
        .slotframeHandle = 0,
        .macLinkIe = {
                        .timeslot = 1,
                        .channelOffset = 0,
                        .macLinkOptions =
                        {
                          .rx = 1,
                          .timekeeping = 1,
                        },
                     },
  },
  {
        .macLinkHandle = 1,
        .macNodeAddress = mDefaultValueOfCoordAddress_c,
        .macLinkType = gMacLinkTypeNormal_c,
        .slotframeHandle = 0,
        .macLinkIe = {
                        .timeslot = 2,
                        .channelOffset = 0,
                        .macLinkOptions =
                        {
                          .tx = 1,
                        },
                     },
  },
};
```

The TSCH primitives configuration is shown in this example:

Example 22. TSCH configuration on an end-device

```
static uint8_t App_Configure_MAC_TSCH_Params(void)
{
    uint32_t i;
    uint16_t hoppingSequenceLength = mMacHoppingSequenceLen_c;
    uint8_t mHoppingSequenceList[] = mMacHoppingSequenceList_c;
    macTschRole_t tschRole = gMacTschRoleDevice_c;

    /* Message for the MLME will be allocated and attached to this pointer */
    mlmeMessage_t *pMsg = MSG_AllocType(mlmeMessage_t);

    if( pMsg != NULL )
    {
        /* Configure addressing PIBs */
        /* MAC short address */
        pMsg->msgType = gMlmeSetReq_c;
        pMsg->msgData.setReq.pibAttribute = gMPibShortAddress_c;
        pMsg->msgData.setReq.pibAttributeValue = (uint16_t*)&macShortAddress;
        (void)NWK_MLME_SapHandler( pMsg, mMacInstance );

        /* Set MAC Hopping Sequence Length */
        pMsg->msgType = gMlmeSetReq_c;
        pMsg->msgData.setReq.pibAttribute = gMPibHoppingSequenceLength_c;
        pMsg->msgData.setReq.pibAttributeValue = &hoppingSequenceLength;
        (void)NWK_MLME_SapHandler( pMsg, mMacInstance );

        /* Set MAC Hopping Sequence List */
        pMsg->msgType = gMlmeSetReq_c;
        pMsg->msgData.setReq.pibAttribute = gMPibHoppingSequenceList_c;
        pMsg->msgData.setReq.pibAttributeValue = mHoppingSequenceList;
        (void)NWK_MLME_SapHandler( pMsg, mMacInstance );

        /* Set slotframes */
        for( i=0; i<sizeof(slotframeArray)/sizeof(macSlotframeIe_t); i++ )
        {
            pMsg->msgType = gMlmeSetSlotframeReq_c;
            pMsg->msgData.setSlotframeReq.operation = gMacSetSlotframeOpAdd_c;
            pMsg->msgData.setSlotframeReq.slotframeHandle =
slotframeArray[i].macSlotframeHandle;
            pMsg->msgData.setSlotframeReq.size = slotframeArray[i].macSlotframeSize;
            (void)NWK_MLME_SapHandler( pMsg, mMacInstance );
        }

        /* Set links */
        for( i=0; i<sizeof(linkArray)/sizeof(macLink_t); i++ )
        {
            pMsg->msgType = gMlmeSetLinkReq_c;
            pMsg->msgData.setLinkReq.operation = gMacSetLinkOpAdd_c;
            pMsg->msgData.setLinkReq.slotframeHandle = linkArray[i].slotframeHandle;
            pMsg->msgData.setLinkReq.linkHandle = linkArray[i].macLinkHandle;
            pMsg->msgData.setLinkReq.linkType = linkArray[i].macLinkType;
            pMsg->msgData.setLinkReq.nodeAddr = linkArray[i].macNodeAddress;
            pMsg->msgData.setLinkReq.timeslot = linkArray[i].macLinkIe.timeslot;
            pMsg->msgData.setLinkReq.channelOffset = linkArray[i].macLinkIe.channelOffset;
            pMsg->msgData.setLinkReq.linkOptions = linkArray[i].macLinkIe.macLinkOptions;
            (void)NWK_MLME_SapHandler( pMsg, mMacInstance );
        }

        /* Configure PANC as time source neighbor */
        pMsg->msgType = gMlmeKeepAliveReq_c;
        pMsg->msgData.keepAliveReq.dstAddr = mDefaultValueOfCoordAddress_c;
        pMsg->msgData.keepAliveReq.keepAlivePeriod = mMacKeepAlivePeriod_c;
        (void)NWK_MLME_SapHandler( pMsg, mMacInstance );

       /* Set TSCH Role as normal node */
       pMsg->msgType = gMlmeSetReq_c;
       pMsg->msgData.setReq.pibAttribute = gMPibTschRole_c;
       pMsg->msgData.setReq.pibAttributeValue = &tschRole;
       (void)NWK_MLME_SapHandler( pMsg, mMacInstance );
       /* Enable TSCH */
        pMsg->msgType = gMlmeTschModeReq_c;
        pMsg->msgData.tschModeReq.tschMode = gMacTschModeOn_c;
        (void)NWK_MLME_SapHandler( pMsg, mMacInstance );

        /* Free message, all requests were sync */
        MSG_Free(pMsg);
    }
    else
    {
        /* Allocation of a message buffer failed. */
        return errorAllocFailed;
    }

    return errorNoError;
}
```

6.20.5. TSCH limitations

Association feature is not available when the TSCH is enabled. Data transmission with extended address mode is not available when TSCH is enabled, because the node address of a link can only support short address mode.

7. Interfacing 802.15.4 MAC Black Box through FSCI

Due to code size or RAM usage considerations, the application and/or network layers cannot run on the same board together with the MAC and PHY layers. For this purpose, the MAC layer can be interfaced by the application and/or network layer as a black box through a serial interface using the FSCI protocol.

The application and/or network layer are denoted as a FSCI Host for interfacing the MAC FSCI Application. Support for this feature is provided without modifying the MAC interface, that is MAC SAP API, NWK SAP registration, and so on. The MAC upper layer must register the serial interface through which the MAC instance is accessed.

The MAC SAPs are accessed in the same manner as before (MAC to NWK), as well as NWK to MAC messages remain unchanged.

The FSCI Host support for MAC module is responsible for providing the NWK layer with MAC SAPs that handle the NWK to MAC messages. Any NWK message is parsed and serialized to be sent as a FSCI packet to the FSCI serial interface registered for the MAC instance. When a FSCI packet is received from the MAC Black Box, it is parsed and a MAC to NWK message is formed and sent to the registered NWK layer SAP for that MAC instance.

When NWK layer acts as a FSCI Host, FSCI monitoring in the NWK SAPs is not needed, as the monitor is performed in the FSCI Host module.

The MAC bind procedure is not performed at MAC init. The FSCI Host must know before how many MAC instances are going to access, and through which serial interfaces.

7.1. FSCI Host for MAC support enablement

7.1.1. Macro definitions

7.1.1.1. gFsciHost_802_15_4_c

This enables the MAC support for a FSCI Host interfacing a MAC black box and enable also the gFsciHostSupport_c:

```
#define gFsciHost_802_15_4_c   0
```

7.1.1.2. gFsciHostSupport_c

This enables the connectivity framework module that is required when interfacing any black box.

7.1.1.3. gFsciHostSyncUseEvent_c

This enables the event mechanism option for the MAC synchronous primitives. It can be enabled only for RTOS environments:

```
#define gFsciHostSyncUseEvent_c 0
```

7.1.1.4. gFsciIncluded_c

This enables the FSCI support for the serial interface with the MAC black box:

```
#define gFsciIncluded_c         0
```

7.1.2. Initialization code

The initialization code remains unchanged for all other components. The MAC layer initialization must be performed to initialize the MAC FSCI Host module. The PHY layer initialization is optional because it does not perform any operation for the FSCI Host.

```
/* FSCI Interface Configuration structure */
static const gFsciSerialConfig_t mFsciSerials[] = {
    /* Baudrate,           interface type,            channel No,       virtual interface */
    {APP_SERIAL_INTERFACE_SPEED, APP_SERIAL_INTERFACE_TYPE, APP_SERIAL_INTERFACE_INSTANCE,
0},
};

/* This table contains indexes into the mFsciSerials[] table for the MAC Black Box */
static uint8_t mFsciHostInterface[gMacInstancesCnt_c];
static instanceId_t mMacInstance[gMacInstancesCnt_c];

/* FSCI init */
FSCI_Init( (void*)mFsciSerials );

/* 802.15.4 MAC init */
MAC_Init();

/* Bind to MAC layer */
for( i=0; i<gMacInstancesCnt_c; i++)
{
    mFsciHostInterface[i] = i;
    mMacInstance[i] = BindToMAC((instanceId_t)i);
    Mac_RegisterSapHandlers(MCPS_NWK_SapHandler, MLME_NWK_SapHandler, mMacInstance[i]);
    fsciRegisterMacToHost( mMacInstance[i], mFsciHostInterface[i] );
    fsciRegisterAspToHost(0, mFsciHostInterface[i]);
}
```

7.2. MAC Synchronous primitives

The MAC layer primitives can be either synchronous or asynchronous. For the asynchronous primitives, the NWK to MAC message is sent to the MAC SAP, and the MAC to NWK confirm is sent back to the NWK SAP after request completion. A synchronous primitive (like the MLME-SET.request) does not wait for a confirmation message from the MAC layer, but accesses the result of the request immediately after the SAP call is finished.

To keep this mechanism unmodified, the FSCI Host module must wait inside the SAP for the FSCI packet response to the current request (after serializing a MAC synchronous request as a FSCI packet and sending it to the black box) and fill the information in the NWK request to be accessed immediately after the MAC SAP execution ends.

This can be achieved in two ways:

1. By the calling task (NWK) to block waiting for an event in the RTOS environment.
2. By polling for the serial response—option that can be used in any OS environment.
   The configuration is performed through the gFsciHostSyncUseEvent_c macro definition.

Appendix A. Auto Frequency Correction (AFC)

A.1. AFC description

The MKW01 platform is capable of performing Auto Frequency Correction. This feature ensures that the receiver frequency can be automatically adjusted with an on-the-fly-computed offset to match the actual transmitter frequency, which can be slightly deviated from the designated carrier center frequency. The AFC is automatically performed on the receiver side, when the RSSI value on the receiver bandwidth exceeds the configured RSSI threshold. The AFC can be performed on either packet preamble or on random noise that triggered the procedure to start. After the AFC is performed, the receiver configured frequency is adjusted with an offset based on the frequency error indicator. This effect is persistent until the receiver block restarts and the frequency resets to the initial value.

The fact that the offset is adjusted directly in the receiver operating frequency implies that the receiver must be restarted if the RSSI value that triggers the AFC is generated by random noise. If the receiver block is not restarted, the receiver frequency is shifted with erroneous offset for following packets. The false trigger in case of noise is handled by setting Rx timeout restart. After the RSSI value exceeded the RSSI threshold and AFC has been performed, a timeout is started with the duration of the preamble and synchronization header to be received. If these events do not occur as expected, the receiver block is automatically restarted and the operating frequency resets to its default value. The AFC frequency tune value is configured to be automatically cleared before each new procedure starts. The RSSI threshold level has an important role in the AFC performance. If a very high sensitivity is set, the chance of false trigger is increased, resulting in Rx timeouts leading to poor performance. The AFC function is disabled during the CCA/ED procedure, as the RSSI values can be erroneously measured due to larger receiver bandwidth. If adjacent energy is present at the edges of the channel spacing occupied by the device, it is detected and reported, and AFC is disabled.

A.2. AFC configuration

By default, AFC feature is not enabled. Turn it on by enabling `gAfcEnabled_d` in the `PhyConfig.h` header file.

The main AFC parameters to configure are:

- AfcRxBw—the receiver bandwidth; this is normally larger than default RxBw, but it is set to a value that does not exceed the operating channel spacing for the device.
- RssiThreshold—the receiver sensitivity; this value must be tuned so that the balance between long-range communication and false Rssi trigger is obtained. The AFC performance is degraded if it is performed on random noise instead of actual preamble of packet.
- LowBetaAfcOffset—additional offset to be added to the frequency error indicator value, when tuning the local oscillator frequency.

The receiver bandwidth must be configured to comply with the following rule:

Bitrate < 2 x RxBw < 2 x AfcRxBw

The LowBetaAfcOffset influences the setting of the local oscillator frequency according to this formula:

AfcCorrection = FeiValue + (LowBetaAfcOffset x 448 Hz)

FeiValue is the frequency offset value measured by the AFC block, while LowBetaAfcOffset is the predefined user value. The AfcCorrection value is then used to tune the receiver frequency. Ensure that the LowBetaAfcOffset exceeds the DC canceller’s cutoff frequency, which is set at about 10 % of the RxBw receiver bandwidth. The DC canceller’s cutoff frequency must be set at a value of about 4 % of the RxBw receiver bandwidth, and it is necessary to remove any DC offset generated through self-reception. The AFC settings can be configured at compile time by adjusting each PHY mode parameters using this structure:

```
typedef struct phyRFConstatnts_tag
{
  uint32_t firstChannelFrequency;
  uint32_t channelSpacing;
  uint16_t totalNumChannels;
  uint16_t bitRateReg;
  uint16_t fdevReg;
  uint8_t  rxBwReg;
  uint8_t  rxBwAfcReg;
  uint8_t  modulationParam;
  uint8_t  ccaThreshold;
  uint8_t  rssiThreshold;
  uint8_t  lowBetaAfcOffset;
} phyRFConstants_t;
```

The default values for AFC settings are:

Table 19. AFC settings

PhyMode	PhyMode 1	PhyMode 2	ARIB 1	ARIB 2	ARIB 3
RxBw	(DccFreq_5 | RxBw_83300)	(DccFreq_5 | RxBw_166700)	(DccFreq_5 | RxBw_83300)	(DccFreq_5 | RxBw_166700)	(DccFreq_5 | RxBw_333300)
RxBwAfc	(DccFreq_7 | RxBw_125000)	(DccFreq_7 | RxBw_250000)	(DccFreq_7 | RxBw_125000)	(DccFreq_7 | RxBw_250000)	(DccFreq_7 | RxBw_500000)
RssiThreshold	0xBE	0xBE	0xBE	0xBE	0xB8
LowBetaAfcOffset	0x07	0x0D	0x07	0x0D	0x0D

The designated RF channel center frequency can be adjusted using an offset through `gAdditionalRFCarrierFreqOffset_c`.

This value is expressed in Fstep units (to be set in transceiver register), and it is calculated as follows:

```
Fstep = 57.220458984375  - 30.0 MHz
Fstep = 61.03515625      - 32.0 MHz
Channel RF Value = (channelFreq + gAdditionalRFCarrierFreqOffset_c) * Fstep
```

Appendix B. Low-Power Modes and Low-Energy Features

B.1. Platform low-power modes

MAC Layer can put both the MCU and the transceiver into low-power modes, based on events generated by MAC low-energy features Coordinated Sampled Listening and Receiver-Initiated Transmission. The desired low-power mode is configured at compile time and selected from:

- MCU LLS/Transceiver Sleep Mode
- LLS Mode allows state recovery using LLWU through these wakeup sources:
  - GPIO interrupt (switch)
  - LPTMR interrupt
- MCU VLPS/Transceiver Sleep Mode
- VLPS Mode allows state recovery through these wakeup sources:
  - GPIO interrupt (switch)
  - LPTMR interrupt
  - UART interrupt (RXEDGIF)

The desired low-power mode is configured by setting `cPWR_DeepSleepMode` to a value from one of the above—1 for LLS mode, and 2 for VLPS mode.

B.2. LPTMR configuration

Clock source—LPO (internal low-power oscillator). At PWR module initialization, this clock source is calibrated versus a precise clock source by measuring its period in PhyTime ticks. The PhyTime tick period is represented by 0.(6) microseconds. This value is transparent to the PWR module. Conversion from microseconds to PhyTime ticks (and vice versa) is provided by PhyTime module.

The duration of the startup calibration is configured through `gLPOCalibrationTicks_c`, value is expressed in timer resolution. Default value is 100 milliseconds.

Wakeup timeout—configurable at runtime in two ways:

- Relative time in milliseconds or unlimited period by user application.
- Absolute time in PhyTime ticks by MAC Layer exclusively.

If the MAC wakeup moment differs from the user wakeup moment, the smallest sleep duration is assumed. If you configure an unlimited period as sleep duration and the MAC wakeup moment is in the past, the LPTMR is disabled as a wakeup source.

Resolution—depends on the configured wakeup timeout. The smallest resolution to make the wakeup timeout possible is chosen. The default value of the period is 1 millisecond.

B.3. Low-power mode configuration

B.3.1. User application

User application can configure the low-power module through the following API:

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

Observations:

- deepSleepTimeMs is a relative timeout in milliseconds
- If deepSleepTimeMs is 0, PWR module is can enter sleep for an unlimited period
- deepSleepTimeMs is valid until another call to this function is made

Low-power entry is allowed or disallowed through the following API:

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

B.3.2. MAC Layer

MAC Layer automatically configures the wakeup timeout and enables/disables the device to sleep, based on user configuration of MAC Low-Energy PIBs.

```
void PWR_SetAbsoluteWakeupTimeInPhyTicks
(
phyTime_t phyTicks
);
typedef phyTime_t uint64_t;
```

Observations:

- phyTicks is an absolute time in PhyTime ticks.
- This function is not to be used by user application. It is designed to be called by MAC Layer exclusively.
- Time is represented as a 64-bit absolute time definition composed by concatenation of a 48-bit software component and a 16-bit free-running counter hardware component. This clock is inactive during sleep period.

The same functions are used for enabling or disabling Low-Power Entry:

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

When Low-Power entry is enabled by all parties involved (MAC Layer and user application), the PWR module reads current clock in phyTime ticks, stops the timer, and enters Low Power. At wakeup, the PWR module starts the timer and updates the phyTime ticks value with the sleep duration.
The interaction between the PhyTime and PWR module is performed through this API:

```
void PhyTimeReadClockTicks
(
  phyTime_t *pRetClk
);
void PhyTimeSyncClockTicks
(
phyTime_t ticks
);
```

B.4. MAC layer low-power modes

Network mode	Coordinator			End device		
	When	**MCU**	**Transceiver**	**When**	**MCU**	**Transceiver**
CSL	N/A	N/A	N/A	CSL Rx	VLPS	Sleep Mode
RIT	N/A	N/A	N/A	RIT On	VLPS/LLS	Sleep Mode
RIT IE	N/A	N/A	N/A	RIT IE On	VLPS	Sleep Mode

B.4.1. Coordinated Sampled Listening

CSL channel sampling is performed once every macCSLPeriod, measured in units of 10 MAC symbols on the currently supported PHY Modes: 200 µs / 100 µs / 50 µs. A common use case is a value of 1 second between CSL channel samples.

CSL Rx can be performed for an indefinite period in a NB PAN.

Low-Power Mode is entered on the receiving node after the upper layer issues MLME-SET.req to macCSLPeriod MAC PIB, and these steps are performed:

- A channel sample is performed with the duration of macPhyCCADuration.
- If CSL energy threshold is reached, the CSL data receive procedure continues without any MCU/TRX Low-Power Modes.
- If CSL energy threshold is not reached, the MAC layer computes the absolute time of the next channel sample considering a guard time, and calls the PWR function specifying the absolute time when to wake up.

Low-Power Recovery is performed first of all by the configured wakeup interrupts to occur.

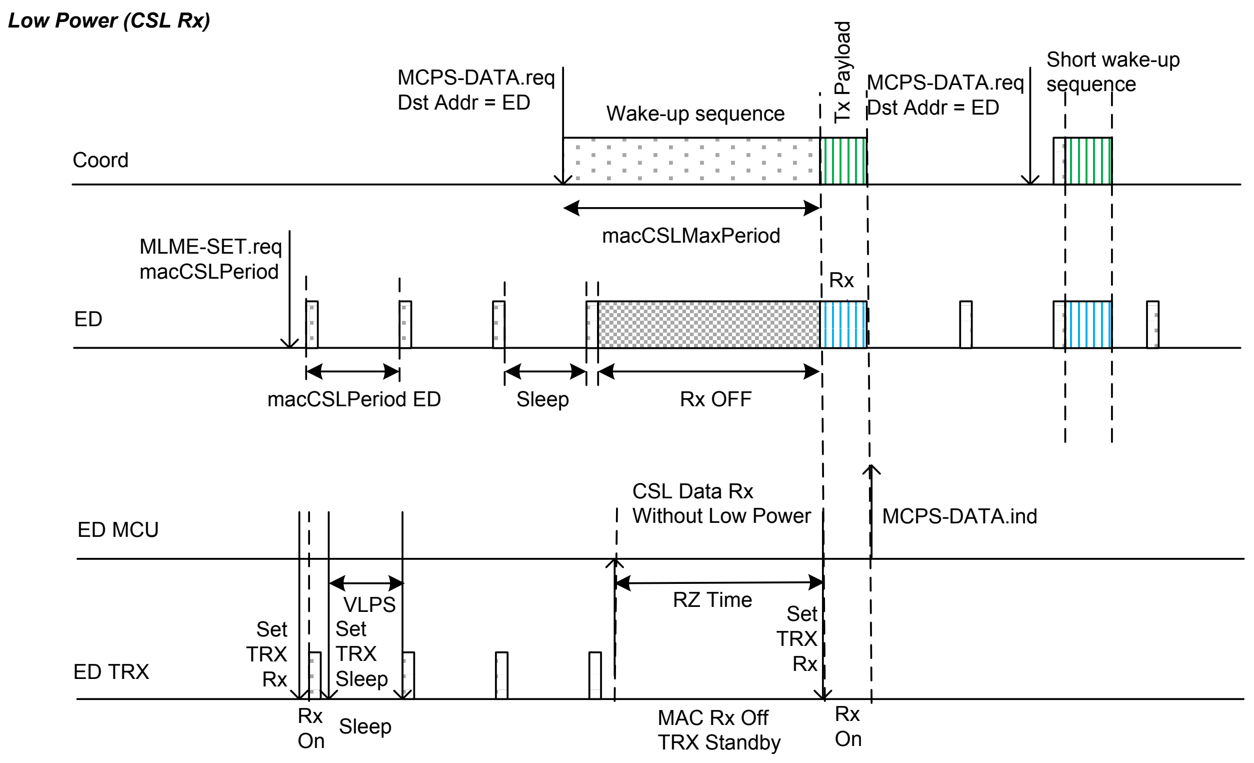

Figure 9. Low-Power CSL Rx

B.4.2. Receiver-Initiated Transmission

Receiver-Initiated Transmission generates a RIT Data Request every macRITPeriod, measured in units of aBaseSuperframeDuration (960 MAC symbols). This value translates using the currently supported PHY modes to 19.2/9.6/4.8 ms. A common usage case is around (or above) 10 seconds, due to the amount of energy required by Tx RIT Data Request operation.

RIT mode is available only in non-beacon mode networks. It can be performed for an indefinite period at the discretion of the upper layer by setting the macRITPeriod PIB to a positive value.

B.4.2.1. RIT without RIT IE listening payload

The receiver device enters Low-Power Mode during the interval starting after the Rx macRITDataWaitDuration until the next RIT Data Request command is sent as follows:

- MAC layer computes the absolute time of the next RIT Data Request command transmission (macRITPeriod - macRITDataWaitDuration) considering a Tx RIT Data Request guard time for the event to be scheduled.
- MAC layer calls the PWR function specifying the absolute time when to wake up.

Low-Power Recovery is performed by the first of all configured wakeup interrupts to occur.

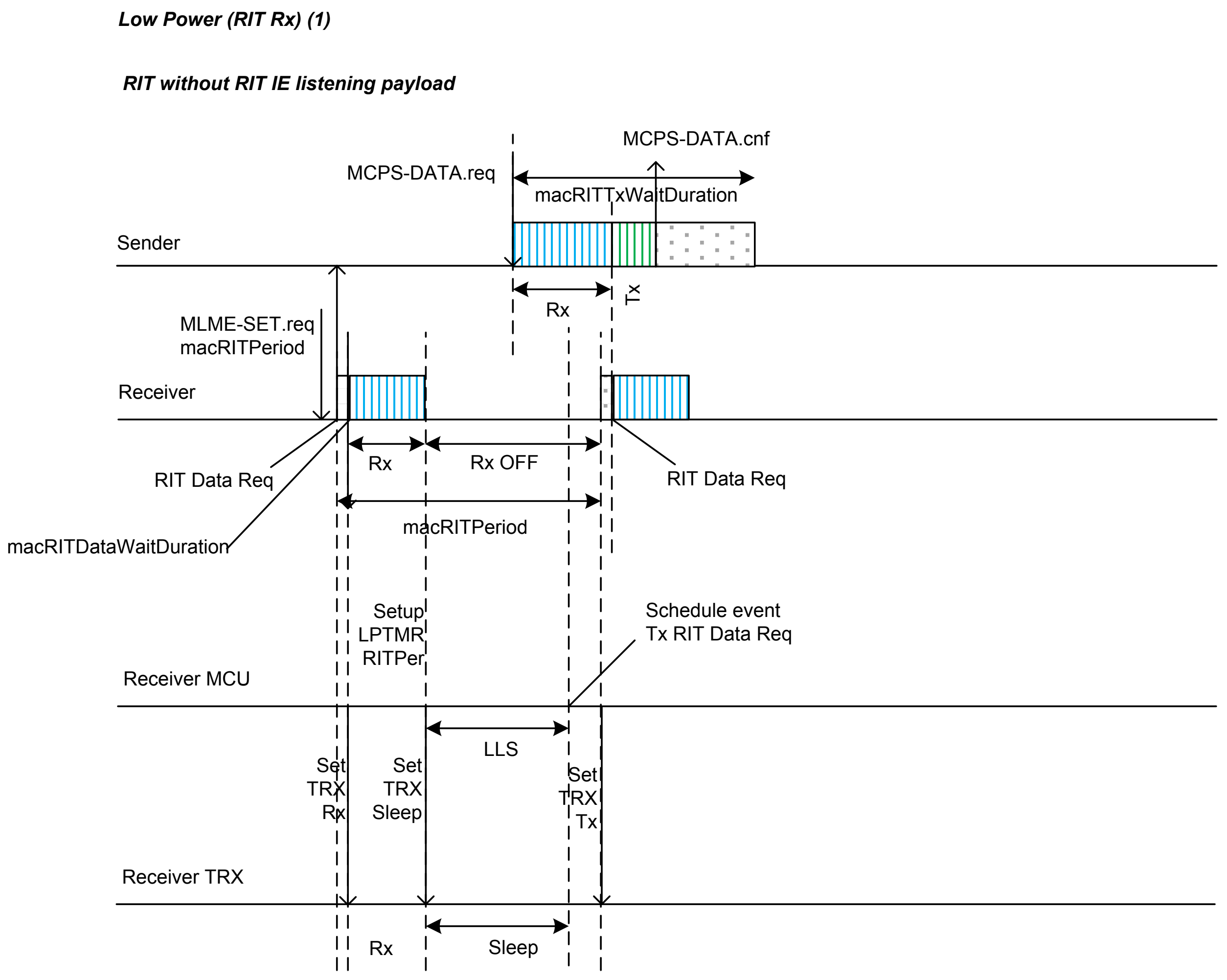

Figure 10. Low-Power RIT Rx (1)

B.4.2.2. RIT with RIT IE listening payload

When RIT IE is included in the RIT Data Request commands, then the receiver performs a listening schedule inside the macRITPeriod interval, as follows:

- It sleeps for the initial T0 period.
- It enables its receiver for macRITDataWaitDuration for N times at T period from each other.

The MAC layer enters Low-Power Mode in one of these states:

- Initial Rx Off
  - MAC layer computes the absolute time of the first Rx On interval after T0, considering the Rx Enable guard time, and calls the PWR module.
- Middle Rx Off
  - MAC layer computes the absolute time of the next Rx On interval after each T, considering the Rx Enable guard time, and calls PWR module.

- Last Rx Off
  - MAC layer computes the absolute time of the next RIT Data Request transmission and Tx guard time, and calls PWR module.

Low-Power Recovery is performed by the first of all configured wakeup interrupts to occur.

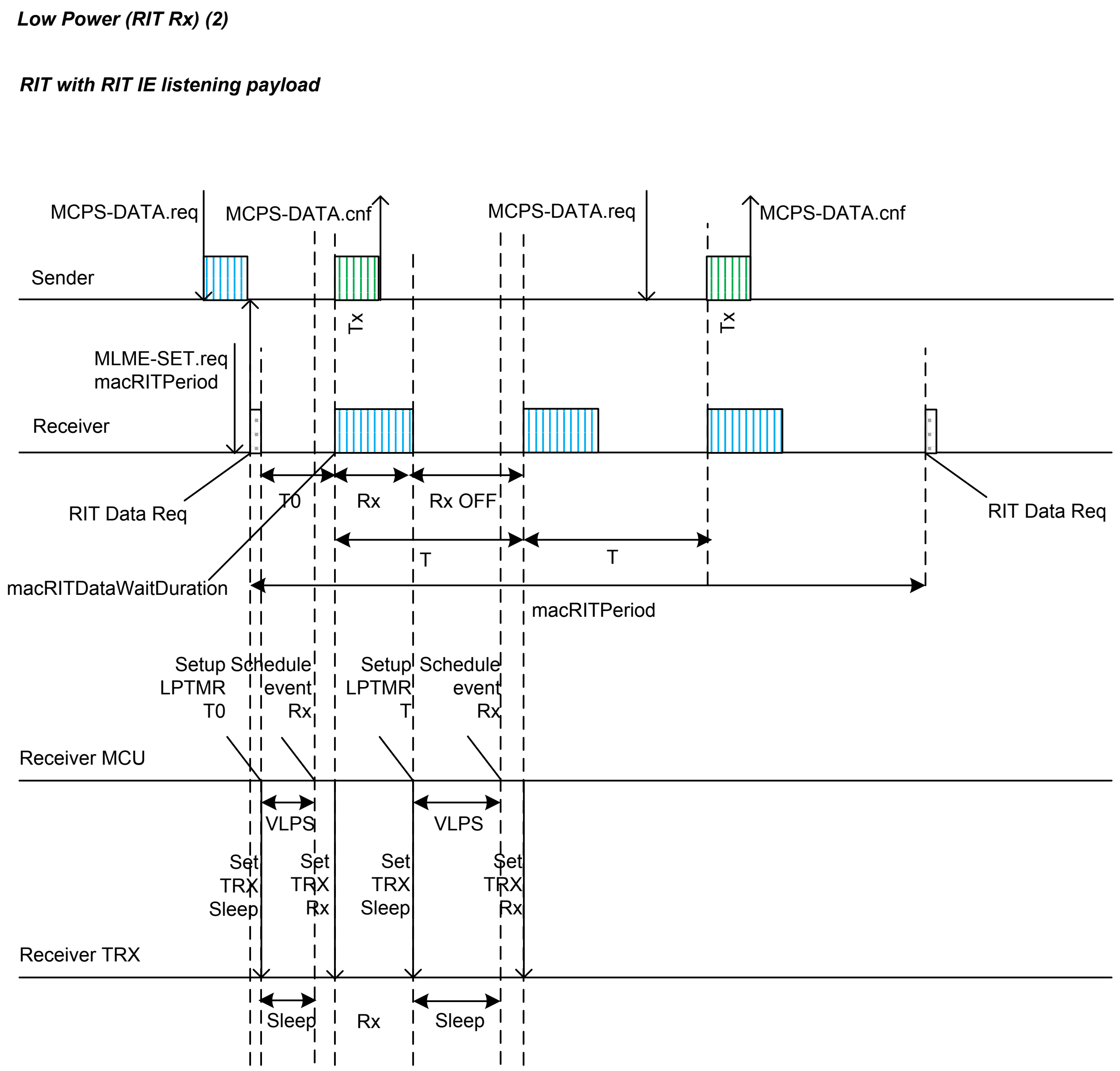

Figure 11. Low-Power RIT Rx (2)

Appendix C. Adding New Sub-GHz PHY Modes on MKW01

C.1. PHY mode structure

PHY Layer implementation includes a subset of the IEEE 802.15.4g FSK PHY modes grouped by Frequency Band Identifiers in a phyPibRFConstants array in *PhyPib.c* source file. Additional non-standard PHY modes can be configured for use. Each new entry of these custom PHY modes must comply with the following structure, as defined in *PhyPib.h* header file:

PHY constants

```
typedef struct phyRFConstatnts_tag
{
  uint32_t firstChannelFrequency;
  uint32_t channelSpacing;
  uint16_t totalNumChannels;
  uint16_t bitRateReg;
  uint16_t fdevReg;
  uint8_t  dccFreq;
  uint8_t  rxFilterBw;
  uint8_t  dccFreqAfc;
  uint8_t  rxFilterBwAfc;
  uint8_t  modulationType;
  uint8_t  modulationShaping;
  uint8_t  ccaThreshold;
  uint8_t  rssiThreshold;
  uint8_t  lowBetaAfcOffset;
} phyRFConstants_t;
```

The parameters are:

- firstChannelFrequency—channel 0 center frequency
- channelSpacing
- totalNumChannels—maximum supported channel number (n, its range is 0 to n-1)
- bitRateReg—bit rate, as set in XCVR register
- fdevReg—frequency deviation, as set in XCVR register
- dccFreq—cut-off frequency of the DC offset canceller
- rxFilterBw—single-side channel filter bandwidth
- dccFreqAfc—cut-off frequency of the DC offset canceller (when AFC is enabled)
- rxFilterBwAfc—single-side channel filter bandwidth (when AFC is enabled)
- modulationType—FSK
- modulationShaping—Gaussian filter BT value applied

- ccaThreshold—threshold (in dBm) to report medium access as busy
- rssiThreshold—receiver sensitivity (when AFC is enabled)
- lowBetaAfcOffset—additional offset when correcting frequency error indicator (when AFC is enabled)

Here is an example of US 902-928 MHz Frequency Band ID FSK PHY Mode 1, as described by IEEE 802.15.4g Standard:

US 902-928 MHz Frequency Band

```
// 902 Operating Mode 1
  {
  //channel center frequency 0
    902200000,
  //channel spacing
    200000,
  //total number of channels
    129,
  //Radio bit rate register
    Bitrate_50000,
  //Radio frequency deviation register
    Fdev_25000,
  //DC Cut-off frequency
    DccFreq_5,
  //Radio Filter Bandwidth
    RxBw_83300,
  //DC Cut-off frequency AFC
    DccFreq_7,
  //Radio Filter Bandwidth AFC
    RxBw_125000,
  //Radio modulation type
    DataModul_Modulation_Fsk,
  //Radio modulation shaping
    DataModul_ModulationShaping_NoShaping,
  //CCA threshold
    81,
  //RSSI threshold
    0xBE,
  //Low Beta Afc Offset
    0x07
  },
```

C.2. PHY RF parameters configuration

The available RF parameters can be chosen from the XCVR register header files, they and are listed for a 32 MHz (FXOSC) oscillator board as an example. These are predefined values for current PHY modes, but new ones can be added as described below.

These values are computed using the following parameters:

- FXOSC—32 MHz
- FSTEP—FXOSC / 2^{19} = ~61

Bitrate—user configurable—obtained by dividing the FXOSC to the desired bitrate value

```
#define Bitrate_1200      0x682B
#define Bitrate_4800      0x1A0B
#define Bitrate_10000     0x0C80
#define Bitrate_20000     0x0640
#define Bitrate_38400     0x0341
#define Bitrate_40000     0x0320
#define Bitrate_50000     0x0280
#define Bitrate_100000    0x0140
#define Bitrate_150000    0x00D5
#define Bitrate_200000    0x00A0
```

Frequency Deviation—user configurable—obtained by dividing the desired frequency to the FSTEP

```
#define Fdev_600       0x000A
#define Fdev_1200      0x0014
#define Fdev_2400      0x0027
#define Fdev_2500      0x0029
#define Fdev_5000      0x0052
#define Fdev_10000     0x00A4
#define Fdev_25000     0x019A
#define Fdev_35000     0x023D
#define Fdev_37500     0x0268
#define Fdev_50000     0x0333
#define Fdev_50049     0x0334
#define Fdev_100000    0x0666
```

DCC Frequency—fixed value domain—set at 0.5 % of the RxBw

```
#define DccFreq_0   (0x00 << 5)
#define DccFreq_1   (0x01 << 5)
#define DccFreq_2   (0x02 << 5)
#define DccFreq_3   (0x03 << 5)
```

```
#define DccFreq_4   (0x04 << 5)
#define DccFreq_5   (0x05 << 5)
#define DccFreq_6   (0x06 << 5)
#define DccFreq_7   (0x07 << 5)
```

Receiver Bandwidth—fixed value domain—set usually at the next available value greater than a multiple of 3 of Frequency Deviation value

```
#define RxBw_2600   ( (RxBwExp_7) | (RxBwMant_2) )
#define RxBw_3100   ( (RxBwExp_7) | (RxBwMant_1) )
#define RxBw_3900   ( (RxBwExp_7) | (RxBwMant_0) )

#define RxBw_5200   ( (RxBwExp_6) | (RxBwMant_2) )
#define RxBw_6300   ( (RxBwExp_6) | (RxBwMant_1) )
#define RxBw_7800   ( (RxBwExp_6) | (RxBwMant_0) )

#define RxBw_10400  ( (RxBwExp_5) | (RxBwMant_2) )
#define RxBw_12500  ( (RxBwExp_5) | (RxBwMant_1) )
#define RxBw_15600  ( (RxBwExp_5) | (RxBwMant_0) )

#define RxBw_20800  ( (RxBwExp_4) | (RxBwMant_2) )
#define RxBw_25000  ( (RxBwExp_4) | (RxBwMant_1) )
#define RxBw_31300  ( (RxBwExp_4) | (RxBwMant_0) )

#define RxBw_41700  ( (RxBwExp_3) | (RxBwMant_2) )
#define RxBw_50000  ( (RxBwExp_3) | (RxBwMant_1) )
#define RxBw_62500  ( (RxBwExp_3) | (RxBwMant_0) )

#define RxBw_83300  ( (RxBwExp_2) | (RxBwMant_2) )
#define RxBw_100000 ( (RxBwExp_2) | (RxBwMant_1) )
#define RxBw_125000 ( (RxBwExp_2) | (RxBwMant_0) )

#define RxBw_166700 ( (RxBwExp_1) | (RxBwMant_2) )
#define RxBw_200000 ( (RxBwExp_1) | (RxBwMant_1) )
#define RxBw_250000 ( (RxBwExp_1) | (RxBwMant_0) )

#define RxBw_333300 ( (RxBwExp_0) | (RxBwMant_2) )
#define RxBw_400000 ( (RxBwExp_0) | (RxBwMant_1) )
#define RxBw_500000 ( (RxBwExp_0) | (RxBwMant_0) )
```

DCC Frequency during AFC—set at 0.125 % of the RxBw when AFC is enabled

```
#define DataModul_ModulationShaping_NoShaping  (0x00 << 0)
```

Receiver Bandwidth during AFC—set at a value greater than RxBw for improved performance when AFC is enabled

```
#define DataModul_ModulationShaping_BT_1       (0x01 << 0)
```

Data Modulation—FSK mode support

```
#define DataModul_ModulationShaping_BT_05      (0x02 << 0)
```

Data Modulation Shaping—set if Gaussian filter is used

```
#define DataModul_ModulationShaping_BT_03      (0x03 << 0)
```

- CCA threshold—set by 802.15.4g Standard at a designated level above the reference sensitivity that all devices must comply with.
- RSSI threshold—receiver sensitivity during AFC, value is set for optimum performance for PHY mode RF parameters.
- Low Beta AFC Offset—FEI additional offset during AFC, value is set for optimum performance for PHY mode RF parameters.

C.3. PHY layer considerations

In most cases, the introduction of a new PHY Mode requires that a new Frequency Band Identifier is introduced. The PHY layer implementation offers only one Frequency Band and corresponding PHY Modes support at a certain moment of time. This is configured from the *PhyTypes.h* header file at compile time.

NOTE

Consider the type of board on which the new frequency band is designed to operate, and configure the inclusion of XCVR register header files accordingly. An important factor in the PHY layer transmission and reception timing is the symbol duration of a PHY Mode. Depending on the bitrate of the desired PHY mode, multiple values can be obtained, for example:

- 1.2 kbps—833.(3) µs symbol
- 20 kbps—50 µs symbol
- 50 kbps—20 µs symbol
- 100 kbps—10 µs symbol
- 150 kbps—6.(6) µs symbol
- 200 kbps—5 µs symbol

Because the PHY layer timings are based on 0.(6) µs timer tick, the symbol conversion must be done accordingly (see the *PhyTime.h* header file for examples).

When preforming transmission or reception, consider the transceiver warm-up times. The warm-up times vary with the bitrate or time to send a bit OTA, and they must be measured according to the desired PHY configuration. You can do this by monitoring the XCVR TxReady or RxReady flags, after issuing a request for Tx or Rx modes. See *MKW01xxRM - MKW01Z128 Reference Manual* (document MKW01XXRM) for more details about these timings and procedures.

Configure the channel spacing in the following function, and see the examples presented there to compute new values using the 30/32 MHz computed FSTEP:

```
void PhyPib_RFUpdateRFfrequency(void)
```

The channel frequency is a multiple of FSTEP, so the channel 0 center frequency and the channel spacing must be calculated with as much approximation as possible.

When PHY Mode is designed to be used with a MAC layer on top, provide the turnaround time (constant 1 ms, expressed in MAC symbols), the PHY maximum channel number, and the scan channel masks, according to the PHY structure.

8. Revision History

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

Table 20. Revision history

Rev. number	Date	Substantive changes
0	07/2015	Initial release
1	08/2015	Changes related to IEEE 802.15.4g/e-specific new features
2	09/2015	Added FSCI host development section
3	04/2016	Added support for KW4x. Updated MAC and PHY API and examples. Updated MAC library naming conventions.
4	09/2016	Public information

How to Reach Us:

Home Page:
nxp.com

Web Support:
nxp.com/support

Document Number: 802154MPADG
Rev. 4
09/2016