802.15.4 MAC/PHY Software

Reference Manual

802154MPSRM/D

Rev. 1.1, 11/2004

Launched by Motorola

freescale™ semiconductor

How to Reach Us:

USA/Europe/Locations Not Listed:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-521-6274 or 480-768-2130

Japan:
Freescale Semiconductor Japan Ltd.
Technical Information Center
3-20-1, Minami-Azabu, Minato-ku
Tokyo 106-8573, Japan
81-3-3440-3569

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
852-26668334

Home Page:
www.freescale.com

Contents

About This Book ..... **viii**

Important Related Documentation ..... **viii**

Organization viii

Conventions viii

Definitions, Acronyms, and Abbreviations ix

References x

Revision History x

Chapter 1 Freescale 802.15.4 MAC/PHY Software Overview **1-1**

1.1 Freescale 802.15.4 MAC/PHY Software Device Types and Libraries 1-1

1.1.1 Full Function Device (FFD) Type 1-2

1.1.2 Full Function Device With No GTS (FFDNGTS) Type 1-2

1.1.3 Full Function Device No Beacon (FFDNB) Type 1-2

1.1.4 Full Function Device No Beacon No Security (FFDNBNS) Type 1-2

1.1.5 Reduced Function Device (RFD) Type 1-2

1.1.6 Reduced Function Device No Beacon (RFDNB) Type 1-3

1.1.7 Reduced Function Device No Beacon No Security (RFDNBNS) Type 1-3

1.1.8 802.15.4_PHY_RD01.Lib 1-3

1.1.9 802.15.4_PHY_AXM_0308_C.Lib 1-3

1.2 The Freescale 802.15.4 MAC/PHY Software Build Environment 1-3

1.2.1 Adding User Applications to the Build Environment 1-4

1.3 Freescale 802.15.4 MAC/PHY Software Source File Structure 1-4

1.4 Configuring the Freescale 802.15.4 MAC/PHY Software (Users Hardware Platform) .. 1-6

1.4.1 Redefining the HCS08 Clock Speed 1-6

1.4.2 Changing the Interconnection between the HCS08 MCU and the MC13192 1-7

Chapter 2 MAC/Network Layer Interface Description **2-1**

2.1 General MAC/Network Interface Information 2-1

2.2 Data Types 2-4

Chapter 3 Feature Descriptions **3-1**

3.1 Configuration 3-1
3.1.1 PIB Attributes 3-1
3.1.2 Configuration Primitives 3-3
3.1.2.1 Reset Request 3-3
3.1.2.2 Reset Confirm (N/A) 3-3
3.1.2.3 Set Request 3-3
3.1.2.4 Set Confirm 3-4
3.1.2.5 Get Request 3-4
3.1.2.6 Get Confirm 3-4
3.1.3 Configuration Examples 3-5
3.2 Scan Feature 3-6
3.2.1 Common Parts 3-6
3.2.2 Energy Detection Scan 3-6
3.2.3 Active and Passive Scan 3-6
3.2.4 Orphan Scan 3-6
3.2.5 Scan Primitives 3-7
3.2.5.1 Scan Request 3-7
3.2.5.2 Scan-Confirm 3-7
3.2.5.3 Orphan Indication 3-8
3.2.5.4 Orphan Response 3-8
3.2.5.5 Beacon Notify Indication 3-8
3.2.5.6 PAN Descriptor 3-9
3.3 Start Feature 3-9
3.3.1 Start Primitives 3-9
3.3.1.1 Start Request 3-9
3.3.1.2 Start Confirm 3-10
3.4 Sync Feature 3-10
3.4.1 Synchronization Primitives 3-10
3.4.1.1 Sync Request 3-10
3.4.1.2 Sync Loss Indication 3-10
3.5 Association Feature 3-10
3.5.1 Association Primitives 3-13

3.5.1.1 Associate Request 3-13
3.5.1.2 Associate Response 3-13
3.5.1.3 Associate Indication 3-13
3.5.1.4 Associate Confirm 3-13
3.5.2 Associate Example 3-14
3.6 Disassociation Feature 3-15
3.6.1 Disassociation Primitives 3-15
3.6.1.1 Disassociate Request 3-15
3.6.1.2 Disassociate Indication 3-16
3.6.1.3 Disassociate Confirm 3-16
3.7 Data Feature 3-16
3.7.1 Data Primitives 3-16
3.7.1.1 Data Request 3-17
3.7.1.2 Data Confirm 3-17
3.7.1.3 Data Indication 3-17
3.7.1.4 Poll Request 3-18
3.7.1.5 Poll Confirm 3-18
3.7.1.6 Communications Status Indication 3-18
3.7.2 Data Example 3-19
3.8 Purge Feature 3-19
3.8.1 Purge Primitives 3-20
3.8.1.1 Purge Request 3-20
3.8.1.2 Purge Confirm 3-20
3.9 Rx Enable Feature 3-20
3.9.1.1 RX Enable Request 3-20
3.9.1.2 RX Enable Confirm 3-21
3.10 Guaranteed Time Slots (GTS) Feature 3-21
3.10.1 GTS as Device 3-21
3.10.2 GTS as PAN Coordinator 3-23
3.10.3 Miscellaneous Items 3-24
3.10.4 GTS Primitives 3-24
3.10.4.1 GTS Request 3-24

3.10.4.2 GTS Confirm 3-24
3.10.4.3 GTS Indication 3-25
3.11 Security 3-25
3.11.1 Security PIB Attributes 3-26
Chapter 4 APP/ASP Layer Interface Description 4-1
4.1 General APP/ASP Interface Information 4-1
4.2 ASP to APP Interface 4-2
4.2.1 Get Time Confirm 4-2
4.2.2 Get Inactive Time Confirm 4-3
4.2.3 Doze Confirm 4-3
4.2.4 Auto Doze Confirm 4-3
4.2.5 Hibernate Confirm 4-3
4.2.6 Wake Confirm 4-4
4.2.7 Event Confirm 4-4
4.2.8 Trim Confirm 4-4
4.2.9 DDR Confirm 4-4
4.2.10 Port Confirm 4-4
4.2.11 CLKO Confirm 4-5
4.2.12 Set Notify Confirm 4-5
4.2.13 Set Min Doze Time Confirm 4-5
4.2.14 Wake Indication 4-5
4.2.15 Idle Indication 4-6
4.2.16 Inactive Indication 4-6
4.2.17 Event Indication 4-6
4.2.18 ASP to APP message union 4-7
4.2.19 Examples of ASP to APP messages 4-7
4.3 APP to ASP Interface 4-8
4.3.1 Get Time Request 4-8
4.3.2 Get Inactive Time Request 4-8
4.3.3 Doze Request 4-9
4.3.4 Auto Doze Request 4-9
4.3.5 Hibernate Request 4-9

4.3.6 Wake Request 4-10
4.3.7 Event Request 4-10
4.3.8 Trim Request 4-10
4.3.9 DDR Request 4-10
4.3.10 Port Request 4-11
4.3.11 CLKO Request 4-11
4.3.12 Set Notify Request 4-11
4.3.13 Set Min Doze Time Request 4-12
4.3.14 APP to ASP message union 4-12
4.3.15 Examples of APP to ASP messages 4-13
Chapter 5 Parametric Information 5-1

About This Book

This document is the Reference Manual for the Freescale IEEETM 802.15.4 Standard, MAC/PHY software. The software is designed for the Freescale MC13192 transceiver. The software package, the MC13192, and the HCS08 MCU forms the Freescale 802.15.4 solution.

Important Related Documentation

It is highly recommended that this reference manual is used together with the Freescale 802.15.4 MAC/PHY Software User's Guide. Users can download this guide from the Freescale ZigBee home page www.freescale.com/zigbee.

Organization

This document is organized into five chapters.

Chapter 1 **Freescale 802.15.4 MAC/PHY Software Overview** — This chapter presents the Freescale 802.15.4 MAC/PHY software device types and libraries, build environment, source file structure, and hardware setup.

Chapter 2 **MAC/Network Layer Interface Description**— This chapter describes the MAC/PHY interface for FDD, RFD and their derivatives.

Chapter 3 **Feature Descriptions** — The chapter contains descriptions of the Freescale 802.15.4 MAC/PHY software features, focusing on the implementation specific details of the IEEE 802.15.4 standard.

Chapter 4 **APP/ASP Layer Interface Description** — This section describes the Application (APP) / Application Support Package (ASP) interface.

Chapter 5 **Parametric Information** — This chapter lists the main parametric information for the Freescale 802.15.4 MAC/PHY software.

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.

Definitions, Acronyms, and Abbreviations

The following list defines the abbreviations used in this document.

ACK	Acknowledgement Frame
API	Application Programming Interface
FFD	Full Function Device as specified in the IEEE 802.15.4 standard.
FFDNGTS	An FFD without GTS support.
FFDNB	An FFD without beacon support.
FFDNBNS	An 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	Micro Controllers
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
PIB	PAN Information Base
PSDU	PHY Service Data Unit
RF	Radio Frequencies
RFD	Reduced Function Device as specified in the IEEE 802.15.4 standard.
RFDNB	An RFD without beacon support.
RFDNBNS	An RFD without beacon or security support.
SAP	Service Access Point
SW	Software

References

The following documents were referenced to build this document.

[1] IEEE™ 802.15.4 Standard -2003, Part 14.5: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs), The Institute of Electrical and Electronics Engineers, Inc. October 2003

[2] ZigBee Security Services Specification V.092

Revision History

The following table summarizes revisions to this manual since the previous release (Rev. 1.0).

Revision History

Location	Revision
Entire Document	Multiple updates.

Chapter 1
Freescale 802.15.4 MAC/PHY Software Overview

This chapter describes the Freescale 802.15.4 MAC/PHY device types and libraries, build environment, source file structure, and hardware setup.

1.1 Freescale 802.15.4 MAC/PHY Software Device Types and Libraries

This section describes the suite of Freescale 802.15.4 MAC/PHY software device types and their related libraries. The different 802.15.4 MAC/PHY software device types offer various degrees of code sizes by reducing functionality. Table 1 shows the relationship between each library and excluded functionalities.

Table 1. MC/PHY Software Device Type Functionality

Device Type	Description	MAC Library File Name	Code Size
FFD	Full-blown FFD. Contains all 802.15.4 features including security.	802.15.4_MAC_FFD_Vx.yz.Lib	37kB
FFDNGTS	Same as FFD but no GTS capability.	802.15.4_MAC_FFDNGTS_Vx.yz.Lib	33kB
FFDNB	Same as FFD but no beacon capability.	802.15.4_MAC_FFDNB_Vx.yz.Lib	28kB
FFDNBNS	Same as FFD but no beacon and no security capability.	802.15.4_MAC_FFDNBNS_Vx.yz.Lib	21kB
RFD	Reduced function device. Contains 802.15.4 RFD features.	802.15.4_MAC_RFD_Vx.yz.Lib	29kB
RFDNB	Same as RFD but no beacon capability.	802.15.4_MAC_RFDNB_Vx.yz.Lib	25kB
RFNBNS	Same as RFD but no beacon and no security capability.	802.15.4_MAC_RFDNBNS_Vx.yz.Lib	18kB

The code size, as shown in Table 1, is for the complete Freescale 802.15.4 MAC/PHY software device type (MAC, PHY, MC13192 driver). x.yz is the version number of the Freescale 802.15.4 MAC/PHY software. The MAC is available in library format only because it is independent of the hardware platform for your 802.15.4 application.

The PHY is available in source code format because it is dependent of the hardware platform for your 802.15.4 application. For example, if users want to run the Freescale 802.15.4 MAC/PHY software on their own hardware platform, they may need to change the definition of the connections between the HCS08 MCU and the MC13192. See Section 1.4 for a detailed description of how to make the Freescale 802.15.4 MAC/PHY software run on the user's own

hardware platform. The Freescale 802.15.4 MAC/PHY software includes two PHY libraries targeted for the Freescale Evaluation boards (See Section 1.1.8 and Section 1.1.9).

NOTE

The PHY is independent of the device type.

1.1.1 Full Function Device (FFD) Type

The Freescale 802.15.4 MAC/PHY software FFD type is an IEEE 802.15.4 Standard compliant Full Functional Device with all MAC features included. It can be used in applications that require both device and coordinator functionality such as ZigBee routers.
Users should compile their application with the 802.15.4_MAC_FFD_Vx.yz.Lib library to create a device with FFD capabilities.

1.1.2 Full Function Device With No GTS (FFDNGTS) Type

The Freescale 802.15.4 MAC/PHY software FFDNGTS device type is an IEEE 802.15.4 Standard compliant FFD with the GTS functionality excluded. It can be used in applications that require both device and coordinator functionality such as ZigBee routers. This library cannot be used for applications demanding GTS data transmissions.
Users should compile their application with the 802.15.4_MAC_FFDNGTS_Vx.yz.Lib library to create a device with FFDNGTS capabilities.

1.1.3 Full Function Device No Beacon (FFDNB) Type

The Freescale 802.15.4 MAC/PHY software FFDNB device type is an IEEE 802.15.4 compliant Full Functional Device with the beacon functionality excluded. It can be used in applications that require both device and coordinator functionality such as ZigBee routers. This library cannot be used for creating beaconed networks.
Users should compile their application with the 802.15.4_MAC_FFDNB_Vx.yz.Lib library to create a device with FFDNB capabilities.

1.1.4 Full Function Device No Beacon No Security (FFDNBNS) Type

The Freescale 802.15.4 MAC/PHY software FFDNBNS device type is an IEEE 802.15.4 compliant Full Functional Device with the beacon and security functionalities excluded. It can be used in applications that require both device and coordinator functionality such as ZigBee routers. This library cannot be used for creating beaconed networks or for applications demanding encrypted or otherwise secured transactions.
Users should link their application with the 802.15.4_MAC_FFDNBNS_Vx.yz.Lib library to create a device with FFDNBNS capabilities.

1.1.5 Reduced Function Device (RFD) Type

The Freescale 802.15.4 MAC/PHY software RFD device type is an IEEE 802.15.4 compliant Reduced Functional Device. It can be used in applications that require device functionality only.

Users should compile their application with the 802.15.4_MAC_RFD_Vx.yz.Lib library to create a device with RFD capabilities.

1.1.6 Reduced Function Device No Beacon (RFDNB) Type

The Freescale 802.15.4 MAC/PHY software RFDNB device type is an IEEE 802.15.4 compliant Reduced Functional Device with the beacon functionality excluded. It can be used in applications that require device functionality only such as leaf devices. That is, end-devices with no child devices. This library cannot be used for applications that need to participate in beaconed networks.
Users should compile their application with the 802.15.4_MAC_RFDNB_Vx.yz.Lib library to create a device with FFDNB capabilities.

1.1.7 Reduced Function Device No Beacon No Security (RFDNBNS) Type

The Freescale 802.15.4 MAC/PHY software RFDNBNS device type is an IEEE 802.15.4 compliant Reduced Functional Device with the beacon and security functionality excluded. It can be used in applications that require device functionality only, such as leaf devices. That is, end-devices with no child devices. This library cannot be used for applications that need to participate in beaconed networks or for applications that require encrypted or otherwise secured transactions.
Users should compile their application with the 802.15.4_MAC_RFDNBNS_Vx.yz.Lib library to create a device with FFDNBNS capabilities.

1.1.8 802.15.4_PHY_RD01.Lib

This Freescale 802.15.4 PHY library is targeted for the Freescale 13192-SARD and 13192-EVB evaluation boards. Users can find more information about these evaluation boards on the Freescale ZigBee homepage www.freescale.com/zigbee. The Freescale 802.15.4 PHY library must be linked with one of the Freescale 802.15.4 MAC libraries and the user application software to form an 802.15.4 compliant product.

1.1.9 802.15.4_PHY_AXM_0308_C.Lib

This Freescale 802.15.4 PHY library is targeted for the Axiom AXM-0308, HCS08 development board. The Freescale 802.15.4 PHY library must be linked with one of the Freescale 802.15.4 MAC libraries and the user application software to form an 802.15.4 compliant product.

1.2 The Freescale 802.15.4 MAC/PHY Software Build Environment

This section describes the Freescale 802.15.4 MAC/PHY software build environment. The Freescale 802.15.4 MAC/PHY software is built using the Metrowerks IDE CodeWarrior Development Studio for Freescale HC08 3.0, build 030506. Freescale recommends using this

version or 3.1, build 4210 of the CodeWarrior tool to build user applications. Both of these versions support the HCS08 variant used in the Freescale 802.15.4 MAC/PHY software package. Two Metrowerks IDE CodeWarrior Development Studio project files are included in this release.

1. **Freescale_802.15.4_PHY_SW_Vx.yz_MC_V30.mcp**. Use this .mcp file if you base your development on the Metrowerks IDE CodeWarrior Development Studio for Freescale HC08 3.0, build 030506.

2. **Freescale_802.15.4_PHY_SW_Vx.yz_MC_V31.mcp**. Use this .mcp file if you base your development on the Metrowerks IDE CodeWarrior Development Studio for Freescale HC08 3.1, build 4210.

1.2.1 Adding User Applications to the Build Environment

This Freescale 802.15.4 MAC/PHY software includes the Freescale 802.15.4 MAC libraries, the Freescale 802.15.4 PHY library/source code, and Metrowerks CodeWarrier project files (.mcp) only. No application library, code, or documentation is included in this release. Adding a user application directly on top of the build environment is possible but it requires both in depth 802.15.4 knowledge and wireless application experience. Freescale strongly recommends that users base their application development on the Freescale 802.15.4 MAC/PHY My_Wireless_App_demo application example software. This software is described in detail in the Freescale *802.15.4 MAC/PHY Software User's Guide*, 802154MPSUG/D. Users can download both the Freescale 802.15.4 My_Wireless_App_demo application example software and the Freescale *802.15.4 MAC/PHY Software User's Guide* on the Freescale ZigBee home page www.freescale.com/zigbee.

1.3 Freescale 802.15.4 MAC/PHY Software Source File Structure

This section describes the source file structure of the Freescale 802.15.4 MAC/PHY software. The Freescale 802.15.4 MAC/PHY software uses the following file extensions:

Source code	*.c *.h
Libraries	*.lib
ELF format targets	*.elf
S19 record format targets	*.s19
Memory maps	*.map

NOTE

All targets are drive and main directory independent. The .mcp-project file and the MAC/PHY libraries will, in a released version,

have a version number added to the end of the file name for easy tracking of versions.

The Freescale 802.15.4 MAC/PHY software is arranged in the following file structure:

```
|——mcp        Metrowerks project file
|——Bin Empty output directory
|——Freescale_Reference_Libs  Freescale Library Files
|   |——Mac
|   |   |——802.15.4_MAC_FFD_Vx.yz.Lib        MAC FFD device type library
|   |   |——802.15.4_MAC_FFDNGTS_Vx.yz.Lib MAC FFDNGTS device type library
|   |   |——802.15.4_MAC_FFDNB_Vx.yz.Lib     MAC FFDNB device type library
|   |   |——802.15.4_MAC_FFDNBNS_Vx.yz.Lib   MAC FFDNBNS device type library
|   |   |——802.15.4_MAC_RFD_Vx.yz.Lib       MAC RFD device type library
|   |   |——802.15.4_MAC_RFDNB_Vx.yz.Lib     MAC RFDNB device type library
|   |   |——802.15.4_MAC_RFDNBNS_Vx.yz.Lib   MAC RFDNBNS device type library
|   |
|   |——Phy
|       |——802.15.4_PHY_AXM_0308_C_Vx.yz.Lib PHY library for Axiom AXM-0308 board
|       |——802.15.4_PHY_RD01_Vx.yz.Lib  PHY library for MC13192SARD and MC13192EVBs
|
|——Src
    |——Code   Main directory for source code
    |   |——Phy  Freescale 802.15.2 PHY source and header files
    |
    |——Ghdr   Global header files and target configuration
```

1.4 Configuring the Freescale 802.15.4 MAC/PHY Software (Users Hardware Platform)

This section describes how to redefine the HCS08 clock speed and how to change the interconnection between the HCS08 MCU and the MC13192. This enables users to run thery 802.15.4 application on their own hardware platform.

1.4.1 Redefining the HCS08 Clock Speed

By properly configuring the Freescale 802.15.4 MAC/PHY software, it is possible to run the HSC08 MCU at various clock speeds. Freescale recommends adding a compiler define "`Type_XXXX`", where XXXX corresponds to the selected MAC library, when the MAC/PHY libraries are linked with the application software. That is to specify "`Type_FFD`" for MAC FFD library, "`Type_RFD`" for MAC RFD library, etc. The MAC libraries were also built using this `#define` to enable the functionality required for a specific device type.
However, the system clock is not directly controlled by the libraries but by the application. In the Freescale 802.15.4 example application My_Wireless_App_demo described in the Freescale *802.15.4 MAC/PHY Software User's Guide*, the system clock is controlled from the files in the `sys` directory. Freescale recommends copying and reusing the files from these examples. Users can download both the Freescale 802.15.4 example application My_Wireless_App_demo and the Freescale *802.15.4 MAC/PHY Software User's Guide* from the Freescale Zigbee web-site at http.//www.freescale.com/zigbee.
The `define` "`Type_XXXX`" selects a minimum system bus frequency in the `App_Target.h` header file in the `sys` directory for each device type. The application can chose to use a higher system bus frequency, but it is not advised to use a lower one. If users want to use a higher frequency than necessary then they need to define one of the following settings on their project:

```
#define SYSTEM_CLOCK_16MHZ
#define SYSTEM_CLOCK_16_78MHZ
```

The above defines are used in the NV_Data.c file to setup the correct system bus frequency. Note that some MCU baud rates are only available at certain system clock frequencies.

1.4.2 Changing the Interconnection between the HCS08 MCU and the MC13192

The PHY library is compiled with standard interconnections. There is one each for the Freescale 13192-SARD and 13192-EVB evaluation boards and one for the AXIOM AXM-0308, HCS08 development board. See Section 1.1.8 and 1.1.9 for a description of these libraries. If users require a different interconnection for their own hardware platform, the PHY library must be recompiled. For this reason the PHY library is delivered as source code.

In the `target.h` header file (in the `Ghdr` directory) a set of macros is defined which are used directly by the PHY layer for antenna control, etc. In addition to the macros used directly by the PHY layer, the GPIO ports on the MCU must be set correctly. The port settings are controlled by an additional set of macros which are also configured in the `target.h` file. The PHY function `PHY_HW_Setup()` uses these macros to set up the ports. Therefore `PHY_HW_Setup()` must be called before calling the `InitializePhy()` function.

The macros in `target.h` can easily be changed for a new hardware configuration. All macros use the following definition which must be redefined if other port and pin mappings are used. In the following example code, the settings are for the Freescale 13192-SARD and 13192-EVB evaluation boards.

```
// Define HW pin mappings
#define ABEL_PORT1          PTCD
#define ABEL_ATT_PIN          (1<<2)
#define ABEL_RxTx_PIN         (1<<3)
#define ABEL_RESET_PIN        (1<<4)

#define ABEL_PORT2          PTBD
#define ABEL_GPIO1_PIN      (1<<4)
#define ABEL_GPIO2_PIN      (1<<5)
#define ABEL_ANT_SWITCH_PIN (1<<6)
```

Chapter 2
MAC/Network Layer Interface Description

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

2.1 General MAC/Network Interface Information

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

1. `zbErrorCode_t MLME_NWK_SapHandler(void *msg)`; MLME to NWK SAP `MLME_NWK_SapHandler()` function passes primitives from the MLME to the NWK)

2. `zbErrorCode_t MCPS_NWK_SapHandler(void *msg)`; MCPS to NWK SAP `MCPS_NWK_SapHandler()` 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.
The SAP handler functions should not be called directly, but through the available macro `MSG_Send(SAP, msg)`. The identifier 'SAP' will be concatenated with `_SapHandler`. Thus, `MSG_Send(NWK_MLME, msg)` will be translated to `NWK_MLME_SapHandler(msg)` where `msg` is some message that must be sent from the NWK to the MLME. Both MLME and MCPS service primitives use the same type of messages as defined in the interface header file called `NwkMacInterface.h`. The macros are defined in the header file called `PhyMacMsg.h`.
The `NWK_MLME_SapHandler()` and `NWK_MCPS_SapHandler()` functions may place a message in a queue. In order to process queued messages, the MLME main function `Mlme_Main()` must be polled from the NWK or from your application.

```
bool _tMlme_Main(void);
```

The function returns `TRUE` if it has more to process (that is, it must be called again) and returns `FALSE` if it does not have more to process. The CPU sleep mode can be entered by the NWK or application.
Because the NWK and MLME/MCPS interfaces are based on messages being passed to a few SAP's, each message needs to have an identifier. These identifiers are shown in the following four tables. Some of the identifiers are unsupported for some of the device types. For example, the MLME-GTS.request primitive is available for the FFDNGTS but the functionality is not supported.

Table 2 provides a list of all the message identifiers in the direction from the MLME to the NWK. They cover all the MLME confirm and indication primitives.

Table 2. Primitives in the MLME to NWK direction.

Message Identifier (primMlmeToNwk_t)	802.15.4 MLME to NWK Primitives
gNwkAssociateInd_c	MLME-ASSOCIATE.Indication
gNwkAssociateCnf_c	MLME-ASSOCIATE.Confirm
gNwkDisassociateInd_c	MLME-DISASSOCIATE.Indication
gNwkDisassociateCnf_c	MLME-DISASSOCIATE.Confirm
gNwkBeaconNotifyInd_c	MLME-BEACON-NOTIFY.Indication
gNwkGetCnf_c	N/A
gNwkGtsInd_c	MLME-GTS.Indication
gNwkGtsCnf_c	MLME-GTS.Confirm
gNwkOrphanInd_c	MLME-RESET.Confirm
gNwkResetCnf_c	N/A
gNwkRxEnableCnf_c	MLME-RX-ENABLE.Confirm
gNwkScanCnf_c	MLME-SCAN.Confirm
gNwkCommStatusInd_c	MLME-COMM-STATUS.Indication
gNwkSetCnf_c	N/A
gNwkStartCnf_c	MLME-START.Confirm
gNwkSyncLossInd_c	MLME-SYNC-LOSS.Indication
gNwkPollCnf_c	MLME-POLL.Confirm

Table 3 provides a list of all the message identifiers in the direction from the MCPS to the NWK. They cover all the MCPS confirm and indication primitives.

Table 3. Primitives in the MCPS to NWK direction.

Message Identifier (primMcpsToNwk_t)	802.15.4 MCPS to NWK Primitives
gMcpsDataCnf_c	MCPS-DATA.Confirm
gMcpsDataInd_c	MCPS-DATA.Confirm
gMcpsPurgeCnf_c	MCPS-PURGE.Confirm

Table 4 provides a list of all the message identifiers in the direction from the NWK to the MLME. They cover all the MLME request and response primitives.

Table 4. Primitives in the NWK to MLME direction.

Message Identifier (primNwkToMlme_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
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

Table 5 provides a list of all the message identifiers in the direction from the NWK to the MCPS. They cover all the MCPS request and response primitives.

Table 5. Primitives in the NWK to MCPS direction.

Message Identifier (primNwkToMcps_t)	802.15.4 NWK to MCPS Primitives
gMcpsDataReq_c	MCPS-DATA.Request
gMcpsPurgeReq_c	MCPS-PURGE.Request

2.2 Data Types

This section describes the main C-structures, and data types used by the MAC/NWK interface. A common feature of all the interface structures is that all elements of a size greater than 1 byte are little endian, and declared as byte arrays. That is, a 16 bit short must be stored as shown in the following code example:

```
short panId = 0x1234;
associateReq->coordPanId[0] = panId & 0xFF; // 0x34
associateReq->coordPanId[1] = panId >> 8;   // 0x12
```

The exception from the little endian notation is the pointer type which may 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 values 0, 2, and 3 meaning No, Short, and Extended Address respectively. In a later release, the definitions for the various values and bit fields will be provided in the `NwkMacInterface.h` header file.
The structures described in Section 3.1.2.1 through Section 3.10.4.3 have been collected in single message type as unions plus a message type that corresponds to the enumeration of the primitives. These are the structures which transport messages across the interface.
For messages from the MLME to the NWK the following structure/union is used.

```
// MLME to NWK message
typedef struct nwkMessage_tag {
  primMlmeToNwk_t msgType;
  union {
    nwkAssociateInd_t     associateInd;
    nwkAssociateCnf_t     associateCnf;
    nwkDisassociateInd_t  disassociateInd;
    nwkDisassociateCnf_t  disassociateCnf;
    nwkBeaconNotifyInd_t  beaconNotifyInd;
    nwkGetCnf_t           getCnf;   // Not used
    nwkGtsInd_t           gtsInd;
    nwkGtsCnf_t           gtsCnf;
    nwkOrphanInd_t        orphanInd;
    nwkResetCnf_t         resetCnf;       // Not used
    nwkRxEnableCnf_t      rxEnableCnf;
    nwkScanCnf_t          scanCnf;
    nwkCommStatusInd_t    commStatusInd;
    nwkSetCnf_t           setCnf;   // Not used
    nwkStartCnf_t         startCnf;
    nwkSyncLossInd_t      syncLossInd;
    nwkPollCnf_t          pollCnf;
  } msgData;
} nwkMessage_t;
```

For messages from the MCPS to the NWK the following structure/union is used:

```
// MCPS to NWK message
typedef struct mcpsToNwkMessage_tag {
  primMcpsToNwk_t msgType;
  union {
    mcpsDataCnf_t   dataCnf;
    mcpsDataInd_t   dataInd;
    mcpsPurgeCnf_t  purgeCnf;
  } msgData;
} mcpsToNwkMessage_t;
```

The following structure/union is used for messages that must be sent from the NWK to the MLME. An MLME message must be allocated using `MSG_AllocType(mlmeMessage_t)`. The macro returns a pointer to a memory location with a sufficient number of bytes, or `NULL` if the memory pools are exhausted. The `NULL` pointer should be handled in the same way as a confirm message with a status code of `TRANSACTION_OVERFLOW`.
An allocated message that is sent to the MLME will be freed automatically. Pay attention to the comments regarding allocation for the Set, Get, and Reset requests described in Section 3.1.2.

```
// NWK to MLME message
typedef struct mlmeMessage_tag {
  primNwkToMlme_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;
  } msgData;
} mlmeMessage_t;
```

The following structure/union is used for messages that must be sent from the NWK to the MCPS. An MCPS-PURGE.request must be allocated using `MSG_AllocType(nwkToMcpsMessage_t)`, while an MCPS-DATA.request message must be allocated using `MSG_Alloc((sizeof( nwkToMcpsMessage_t)-1)+size)`. Both allocation macros return a pointer to a memory location with a sufficient number of bytes, or `NULL` if the memory pools are exhausted. The `NULL` pointer should be handled in the same way as a confirm message with a status code of `TRANSACTION_OVERFLOW`.
An allocated message that is sent to the MCPS will be freed automatically.

```
// NWK to MCPS message
typedef struct nwkToMcpsMessage_tag {
  primNwkToMcps_t msgType;
  union {
    mcpsDataReq_t   dataReq;
    mcpsPurgeReq_t  purgeReq;
  } msgData;
} nwkToMcpsMessage_t;
```

Chapter 3
Feature Descriptions

The chapter contains descriptions of the Freescale 802.15.4 MAC/PHY software features, focusing on the implementation specific details of the IEEE 802.15.4 standard. Refer to the IEEE 802.15.4 standard for more details on these features.

3.1 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 must be configured by the upper layer. The MAC PIB attributes, and the three primitives which are available for configuring the MAC PIB are described in the following sections.

3.1.1 PIB Attributes

Table 6 shows all the MAC PIB attributes available including Freescale specific additions to the 802.15.4 specific attributes.

Table 6. Available PIB Attributes.

PIB Attribute	Description
Freescale Specific Attributes	
0x20	`gMacRole_c`. Contains the current role of the device: 0x00 = Device, 0x01 = Coordinator, 0x02 = PAN Coordinator.
0x21	`gMacLogicalChannel_c`. Contains the current logical channel (11 to 26).
0x22	`gMacPanCoordinator_c`. TRUE if the device is a PAN Coordinator.
802.15.4 Specific Attributes (see Error! Reference source not found. **for descriptions)**	
0x40	gMacAckWaitDuration_c
0x41	gMacAssociationPermit_c
0x42	gMacAutoRequest_c
0x43	gMacBattLifeExt_c
0x44	gMacBattLifeExtPeriods_c
0x45	gMacBeaconPayload_c
0x46	gMacBeaconPayloadLength_c
0x47	gMacBeaconOrder_c
0x48	gMacBeaconTxTime_c
0x49	gMacBsn_c
0x4A	gMacCoordExtendedAddress_c
0x4B	gMacCoordShortAddress_c
0x4C	gMacDsn_c
0x4D	gMacGtsPermit_c

0x4E	gMacMaxCsmaBackoffs_c
0x4F	gMacMinBe_c
0x50	gMacPanId_c
0x51	gMacPromiscuousMode_c
0x52	gMacRxOnWhenIdle_c
0x53	gMacShortAddress_c
0x54	gMacSuperFrameOrder_c
0x55	gMacTransactionPersistenceTime_c
Security Specific Attributes	
0x70	`gMacAclEntryDescriptorSet_c`. A set of ACL entries each containing information to be used to protect frames between the MAC layer and the specified destination device. Size is 30*N, where N is the number of ACL entry descriptors.
0x71	`gMacAclEntryDescriptorSetSize_c`. The number of entries in the ACL descriptor set. Size is 1 byte.
0x72	`gMacDefaultSecurity_c`. If TRUE, then the device is able to send/receive secured frames to/from devices not listed in the ACL descriptor set. Size is 1 byte.
0x73	`gMacDefaultSecurityMaterialLength_c`. The number of bytes in the ACL Security Material. Size is 1 byte.
0x74	`gMacDefaultSecurityMaterial_c`. The specific security material to be used to protect frames between the MAC and devices not in the ACL descriptor set. Size is 16 bytes.
0x75	`gMacDefaultSecuritySuite_c`. The unique identifier of the security suite to be used to protect frames between the MAC and devices not in the ACL descriptor set. Size is 1 byte.
0x76	`gMacSecurityMode_c`. The identifier of the security mode in use. 0x00 = Unsecured mode, 0x01 = ACL mode, 0x02 is Secured Mode. Size is 1 byte.
Freescale Specific Security Attributes	
0x77	`gMacAclEntryCurrent_c`. Sets which ACL entry is active for access (0 indicates first entry, 1 second entry and so on). Size is 1 byte.
0x78	`gMacAclEntryExtAddress_c`. 64 bit addr of the device in this ACL entry. Size is 8 bytes.
0x79	`gMacAclEntryShortAddress_c`. 16 bit addr of the device in this ACL entry. Size is 2 bytes.
0x7A	`gMacAclEntryPanId_c`. PAN ID of the device in this ACL entry. Size is 2 bytes.
0x7B	`gMacAclEntrySecurityMaterialLength_c`. Number of bytes in 'aclSecurityMaterial' (<=16). Size is 1 byte.
0x7C	`gMacAclEntrySecurityMaterial_c`. Key for protecting frames. Size is 16 bytes.
0x7D	`gMacAclEntrySecuritySuite_c`. Security suite used for the device in this ACL entry. Size is 1 byte.

3.1.2 Configuration Primitives

This section describes the implementation of the configuration related primitives.

3.1.2.1 Reset Request

The internal state of the MAC including the message/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 will be 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 `MSG_Alloc()`, it will **not** be freed by the MLME. No confirm message is generated. Instead the return code from the `MSG_Send()` macro is used as the status code.

```
// Type: gMlmeResetReq_c,
typedef struct mlmeResetReq_tag {
  bool_t   setDefaultPib;
} mlmeResetReq_t;
```

3.1.2.2 Reset Confirm (N/A)

The Reset-Confirm 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: gNwkResetCnf_c,
typedef struct nwkResetCnf_tag {
  uint8_t  status;
} nwkResetCnf_t;
```

3.1.2.3 Set Request

The *MLME-SET.request* is used for modifying parameters in the MAC PIB. See Section 3.1 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 of more 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 `MSG_Alloc()`, it will *not* be freed by the MLME. No confirm message is generated. Instead the return code from the `MSG_Send()` macro is used as the status code. When the Set-Request is used 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.

```
// Type: gMlmeSetReq_c,
typedef struct mlmeSetReq_tag {
  uint8_t pibAttribute;
  uint8_t *pibAttributeValue; // Pointer supplied by NWK
} mlmeSetReq_t;
```

3.1.2.4 Set Confirm

The Set-Confirm is not used because the Set-Request is carried out synchronously. See the Set-Request structure for more information.

```
// Type: gNwkSetCnf_c,
typedef struct nwkSetCnf_tag {
  uint8_t  status;
  uint8_t  pibAttribute;
} nwkSetCnf_t;
```

3.1.2.5 Get Request

The `MLME-GET.request` reads parameters in the MAC PIB. See Table 6 for a list of available PIB attributes.

The Get-Request message contains a pointer to a buffer where data from the MAC PIB will be copied to. The pointer must be supplied by the NWK or APP. Attributes with a size of more 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 `MSG_Alloc()`, it will *not* be freed by the MLME. No confirm message is generated. Instead the return code from the `MSG_Send()` macro is used as the status code.

```
// Type: gMlmeGetReq_c,
typedef struct mlmeGetReq_tag {
  uint8_t  pibAttribute;
  uint8_t  *pibAttributeValue; // Pointer supplied by NWK
} mlmeGetReq_t;
```

3.1.2.6 Get Confirm

The Get-Confirm is not used because the Get-Request is carried out synchronously. See the Get-Request structure for more information.

```
// Type: gNwkGetCnf_c,
typedef struct nwkGetCnf_tag {
  uint8_t  status;
  uint8_t  pibAttribute;
  uint8_t  *pibAttributeValue;
} nwkGetCnf_t;
```

3.1.3 Configuration Examples

The following are examples of sending a configuration messages to the MLME.
Send a Set-Request with `macPanId=0x1234`.

```
uint8_t confirmStatus;
uint8_t bPanId[2] = {0x34, 0x12}; // little endian Pan ID
mlmeMessage_t *Msg = MSG_AllocType(mlmeMessage_t);

Msg->msgData.setReq.attribute = 0x50;
Msg->msgData.setReq.attributeValue = bPanId;
Msg->msgType = gMlmeSetReq_c;

// Calls uint8_t NWK_MLME_SapHandler(void *msg)
confirmStatus = MSG_Send(NWK_MLME, Msg)

// Msg is not deallocated by Get/Set/Reset-Requests.
MSG_Free(Msg);
```

Another way is to use the stack instead of using `MSG_AllocType()` for getting the message buffer (*Only* for Get/Set/Reset Requests).

```
mlmeMessage_t msg;
uint8_t autoRequestFlag = TRUE;

// Set message identifer to MLME-SET.request
msg.msgType = gMlmeSetReq;

// We want to set the PAN ID attribute of the MAC PIB.
msg.msgData.setReq.attribute = 0x50;
msg.msgData.setReq.attributeValue = bPanId;

// Calls uint8_t NWK_MLME_SapHandler(void*msg)
confirmStatus = MSG_Send(NWK_MLME, &msg)

// Set the MAC PIB Auto request flag to TRUE. No need to set
// message identifier again since the Msg is not modified by
// the MSG_Send(NWK_MLME, &msg) call.
msg.msgData.setReq.attribute = 0x42;
msg.msgData.setReq.attributeValue = &autoRequestFlag;
MSG_Send(NWK_MLME, &msg);
```

Example of getting `macBeaconTxTime` using the Get Request.

```
uint8_t txTime[3];

msg.msgData.getReq.attribute = 0x48;
msg.msgData.getReq.attributeValue = txTime;
msg.msgType = gMlmeGetReq_c;

// Calls uint8_t NWK_MLME_SapHandler(void*msg)
confirmStatus = MSG_Send(NWK_MLME, &msg)

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

3.2 Scan Feature

In general, this feature is implemented as described in the IEEE 802.15.4 standard (parts 7.1.11, 7.5.2), so only the differences and additional details are included in this document.

3.2.1 Common Parts

Requesting any of the scan types (using *the MLME-SCAN.request* primitive) will interrupt all other system activity at the MLME layer and below, in accordance with the IEEE 802.15.4 standard. It is the responsibility of the NWK layer to only initiate scanning, when this behavior is acceptable.

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

3.2.2 Energy Detection Scan

Energy Detection Scan is not supported on RFD devices and derivatives. When Energy Detection Scan is requested, the device will measure the energy level on each requested channel until the scan time has elapsed.

The *MLME-SCAN.confirm* primitive will always hold energy detection results from all requested channels, that is, partial responses will never be returned.

The energy levels are measured in ½dBm steps, with 0 corresponding to –80dBm (theoretical minimal value) and 0xA0 (decimal 160) corresponding to 0dBm (theoretical maximal value). Practical tests on the MC13192 development board (version 2.0) indicate the practical theoretical minimal value to be 0x0A (-75dBm) and the maximal value to be 0x82 (-15dBm).

3.2.3 Active and Passive Scan

When Active or Passive Scan is requested, the device will wait for beacons to arrive until the scan time has elapsed. If during this time a valid unique beacon is received, the device will store the result. In this case, or if any other package was received from the air, the device will re-enter 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 Scan is capable of returning up to five (5) results in a single *MLME-SCAN.confirm* primitive. Thus, when five (5) unique (see the 802.15.4 Standard) beacons have been received, the Scan is terminated in accordance with the IEEE 802.15.4 standard, even if all channels have not been scanned to completion.

3.2.4 Orphan Scan

When Orphan Scan is requested, the device will wait for a coordinator realignment command to arrive until the scan time has elapsed. If during this time any other command is received from the air, the device will ignore the command and re-enter Rx mode, as long as there is time for the shortest possible Rx cycle to complete before the complete scan time has elapsed.

If a valid coordinator realignment response is received while performing the Orphan Scan, scanning is immediately terminated in accordance with the IEEE 802.15.4 standard **Error!**

Reference source not found., even if all channels have not been scanned to completion. 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.

3.2.5 Scan Primitives

This section describes the implementation of the Scan related primitives.

3.2.5.1 Scan Request

The Scan-Request message parameters map straightforwardly to the message parameters listed and described in **Error! Reference source not found.**. It must be ensured that `scanChannels` always indicates at least one valid channel, and that channels outside the valid range [11;26] are not indicated. The value 0x07FFF800 corresponds to "*all valid channels*". The valid range for `scanType` is [0;3]. The valid range for `scanDuration` is [0;14].

```
// Type: gMlmeScanReq_c,
typedef struct mlmeScanReq_tag {
  uint8_t  scanType;
  uint8_t  scanChannels[4];
  uint8_t  scanDuration;
} mlmeScanReq_t;
```

3.2.5.2 Scan-Confirm

The Scan-Confirm structure contains a pointer to an array of PAN descriptors or energy levels. See the definition in Section 3.2.5.6. The array must be freed by a call to `MM_Free()` *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.

```
// Type: gNwkScanCnf_c,
typedef struct nwkScanCnf_tag {
  uint8_t  status;
  uint8_t  scanType;
  uint8_t  resultListSize;
  uint8_t  unscannedChannels[4];
  union {
    // Byte array [16]. Must be freed by MM_Free();
    uint8_t *pEnergyDetectList;
    // Array of pan descriptors [5]. Must be freed by MM_Free();
    panDescriptor_t *pPanDescriptorList;
  } resList;
} nwkScanCnf_t;
```

3.2.5.3 Orphan Indication

```
// Type: gNwkOrphanInd_c,
typedef struct nwkOrphanInd_tag {
  uint8_t  orphanAddress[8];
  bool_t   securityUse;
  uint8_t  AclEntry;
} nwkOrphanInd_t;
```

3.2.5.4 Orphan Response

```
// Type: gMlmeOrphanRes_c,
typedef struct mlmeOrphanRes_tag {
  uint8_t  orphanAddress[8];
  uint8_t  shortAddress[2];
  bool_t   securityEnable;
  bool_t   associatedMember;
} mlmeOrphanRes_t;
```

3.2.5.5 Beacon Notify Indication

The *MLME-BEACON-NOTIFY.indication* message is not only received during scan, but may also be received when the device is tracking a beaconing coordinator.
The *MLME-BEACON-NOTIFY.indication* message is special since it contains pointers. `pAddrList` points to the address list which is formatted according to the 802.15.4 Standard, Section 7.2.2.1.6/7. `pPanDescriptor` points to the pan descriptor of the indication message. See the definition in Section 3.2.5.6. `pSdu` is the beacon payload buffer. The `pBufferRoot` pointer is containing the data fields pointed to by the other pointers, and is used for freeing only.

WARNING

The `pBufferRoot` must be freed before freeing the indication message. As shown in this example, `MSG_Free(pBeaconInd->pBufferRoot); MSG_Free(pBeaconInd);` Otherwise, the MAC memory pools will be exhausted after just a few beacons.

```
// Type: gNwkBeaconNotifyInd_c
typedef struct nwkBeaconNotifyInd_tag {
  uint8_t  bsn;
  uint8_t  pendAddrSpec;
  uint8_t  sduLength;
  uint8_t  *pAddrList;
  panDescriptor_t *pPanDescriptor;
  uint8_t  *pSdu;
  uint8_t  *pBufferRoot;
} nwkBeaconNotifyInd_t;
```

3.2.5.6 PAN Descriptor

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

```
typedef struct panDescriptor_tag {
  uint8_t  coordAddress[8];
  uint8_t  coordPanId[2];
  uint8_t  coordAddrMode;
  uint8_t  logicalChannel;
  bool_t   securityUse;
  uint8_t  aclEntry;
  bool_t   securityFailure;
  uint8_t  superFrameSpec[2];
  bool_t   gtsPermit;
  uint8_t  linkQuality;
  uint8_t  timeStamp[3];
} panDescriptor_t;
```

3.3 Start Feature

The start feature is not supported on RFD type devices and derivatives.
According to the IEEE 802.15.4 standard **Error! Reference source not found.** it is necessary to set the `macShortAddress` PIB attribute to any value different from 0xFFFF before using the start feature, otherwise an error code `gNoShortAddress_c` will be returned.
It has been chosen to enable `RxOnWhenIdle` when successfully calling the *MLME-START.request* primitive. Hence the new (PAN) coordinator will start receiving right away.
Also, if any additional *MLME-START.request* primitives are issued in order to change superframe configuration after previously having enabled a beaconed network using *MLME-START.request*, all information regarding GTS (if GTS is being used) will be cleared and GTS must be set up again by the NWK layer.

3.3.1 Start Primitives

This section describes the implementation of the Start related primitives.

3.3.1.1 Start Request

Before sending a Start-Request, the *macShortAddress* must be set to something different from 0xFFFF.

```
// Type: gMlmeStartReq_c,
typedef struct mlmeStartReq_tag {
  uint8_t  panId[2];
  uint8_t  logicalChannel;
  uint8_t  beaconOrder;
  uint8_t  superFrameOrder;
  bool_t   panCoordinator;
  bool_t   batteryLifeExt;
  bool_t   coordRealignment;
  bool_t   securityEnable;
} mlmeStartReq_t;
```

3.3.1.2 Start Confirm

```
// Type: gNwkStartCnf_c,
typedef struct nwkStartCnf_tag {
  uint8_t  status;
} nwkStartCnf_t;
```

3.4 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 way to detect when the synchronization occurs is to set the `macAutoRequest` PIB attribute to `FALSE` prior to executing the *MLME-SYNC.request*. This forces the MAC to send an *MLME-BEACON-NOTIFY.indication* every time a beacon is received. After the first beacon has been received, the `macAutoRequest` PIB attribute can be set to `TRUE` again. If `aMaxLostBeacons` consecutive beacons are lost, the MAC will send an *MLME-SYNC-LOSS.indication*. It is very important to set the `macPANId` PIB attribute to a value different from 0xffff prior to executing *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 IDLE state. Notice that 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.

3.4.1 Synchronization Primitives

This section describes the implementation of the Synchronization related primitives.

3.4.1.1 Sync Request

```
// Type: gMlmeSyncReq_c,
typedef struct mlmeSyncReq_tag {
  uint8_t  logicalChannel;
  bool_t   trackBeacon;
} mlmeSyncReq_t;
```

3.4.1.2 Sync Loss Indication

```
// Type: gNwkSyncLossInd_c,
typedef struct nwkSyncLossInd_tag {
  uint8_t  lossReason;
} nwkSyncLossInd_t;
```

3.5 Association Feature

Association is implemented according to the IEEE 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 two MAC command frames being sent to the coordinator. The first is the association request itself. The second is the data request that is sent after `aResponseWaitTime` symbols. Refer to the IEEE 802.15.4 standard for a description of the association procedure. The following attributes are updated when *MLME-ASSOCIATE.request* is called.

- `macPanId`: This attribute is updated as required by the IEEE 802.15.4 Standard.
- `phyLogicalChannel`: This attribute is updated as required by the IEEE 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 IEEE 802.15.4 Standard. If the extended coordinator address is used in the call it is not possible to update this attribute – the short address of the coordinator is unknown. The IEEE 802.15.4 Standard does not mention this possibility. The implementation will force `macCoordShortAddress` to 0xFFFE if an extended address is used in the call.
- `macShortAddress`: The implementation will force this attribute to 0xFFFF before sending the request to the coordinator. This is the default value following a reset. The attribute is updated because it will ensure that the data request is sent using a long source address. This is the only way to guarantee that the association response can be successfully extracted from the coordinator. Setting `macShortAddress` to 0xFFFF can be considered as a safeguard mechanism. Although this update is not listed in the IEEE 802.15.4 Standard, it should not violate the intention of the IEEE 802.15.4 Standard.

Once these attributes have been updated, a MAC command frame containing the association request is then sent to the coordinator. A timer is started upon successful reception. The timer expires after `aResponseWaitTime` symbols. Note that the timeout value has a different meaning given the scenario used.

- Non-beacon enabled PAN network (`macBeaconOrder` = 15). The timeout value just a simple wait (corresponds to approximately 0.5 sec).
- Beacon-enabled PAN network (`macBeaconOrder` < 15). The same interpretation as above 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. The timeout may in this case extend over several superframes.

The last scenario has some implications that may not be evident. Consider a superframe configuration with macBeaconOrder = 14 and macSuperframeOrder = 0. The CAP will in this example be approximately 900 symbols (depends on the length of the beacon frame). Beacons will be transmitted approximately every 4 minutes. `aResponseWaitTime` is equal to 30.720 symbols. This implies that the timeout will occur after approximately 34 superframes which in the example is more than two hours!

The data request is sent when the timer expires in order to get the association response from the coordinator unless the following occurs.

- The association response has been “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. Note that 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 was being tracked when the association procedure was initiated. If this should happen 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 IEEE 802.15.4 standard **Error! Reference source not found.**. Note that an *MLME-SYNC-LOSS.indication* message will also be 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 frame (the expected being the *MLME-ASSOCIATE.response* packet of course). The following attributes are updated if the associate response frame is received and the status indicates a successful association as shown in the following list.

- `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 this is mentioned in the IEEE 802.15.4 standard **Error! Reference source not found.**. The short address of the coordinator is not present in the response.

Notice also that an *MLME-COMM-STATUS.indication* message is generated on the coordinator when the response has been extracted by the device.

3.5.1 Association Primitives

This section describes the implementation of the Association related primitives.

3.5.1.1 Associate Request

Before sending the Associate-Request primitive, the IEEE 802.15.4 standard **Error! Reference source not found.** states that a Reset-Request must be sent, and an Active or Passive Scan was performed.

```
// Type: gMlmeAssociateReq_c
typedef struct mlmeAssociateReq_tag {
  uint8_t  coordAddress[8];
  uint8_t  coordPanId[2];
  uint8_t  coordAddrMode;
  uint8_t  logicalChannel;
  bool_t   securityEnable;
  uint8_t  capabilityInfo;
} mlmeAssociateReq_t;
```

3.5.1.2 Associate Response

```
// Type: gMlmeAssociateRes_c
typedef struct mlmeAssociateRes_tag {
  uint8_t  deviceAddress[8];
  uint8_t  assocShortAddress[2];
  bool_t   securityEnable;
  uint8_t  status;
} mlmeAssociateRes_t;
```

3.5.1.3 Associate Indication

```
// Type: gNwkAssociateInd_c
typedef struct nwkAssociateInd_tag {
  uint8_t  deviceAddress[8];
  bool_t   securityUse;
  uint8_t  AclEntry;
  uint8_t  capabilityInfo;
} nwkAssociateInd_t;
```

3.5.1.4 Associate Confirm

```
// Type: gNwkAssociateCnf_c
typedef struct nwkAssociateCnf_tag {
  uint8_t  assocShortAddress[2];
  uint8_t  status;
} nwkAssociateCnf_t;
```

3.5.2 Associate Example

This section shows an example of sending an associate-request. Some pseudo-code has been used (the `AssociateFillInParms()`, and `HandleAssocConf()` functions does not exist) in order to simplify the example.

```
// Need to allocate MLME message for this one.
mlmeMessage_t *ReqMsg = MSG_AllocType(mlmeMessage_t);
nwkMessage_t *CnfMsg;

// If ReqMsg==NULL then TRANSACTION_OVERFLOW
// fill in source+destination addresses and capabilities.
AssociateFillInParms(&ReqMsg->msgData.associateReq);
ReqMsg->msgType = gMlmeAssociateReq_c;

// Calls uint8_t NWK_MLME_SapHandler(void*msg)
confirmStatus = MSG_Send(NWK_MLME, ReqMsg)

if(confirmStatus == SUCCESS) {
  // OS call that waits until input arrives in the input queue.
  WaitEvent();

  // Use message system to get the message from the input queue.
  if(MSG_Pending(&nwkQueue)) {
    CnfMsg = MSG_DeQueue(&nwkQueue);

    // Check if it is the correct message
    if(CnfMsg->msgType == gNwkAssociateCnf_c) {
      HandleAssocConf(&CnfMsg->msgData.associateCnf);
    }
  }
  else {
    // Not the message we waited for.
  }
  // ALWAYS remember to free incoming messages.
  MSG_Free(CnfMsg);
}
else {
// MAC failed to initiate association request due to either
// wrong parameters or out of buffers. Msg was freed by the MAC.
}
```

3.6 Disassociation Feature

Disassociation is less complex than association but there is an issue in the IEEE 802.15.4 standard that makes disassociation from a coordinator difficult in a non-beacon enabled PAN network.

- Disassociation from a device: The *MLME-DISASSOCIATE.request* is just sent to the remote device where it will result in an *MLME-DISASSOCIATE.indication* message.
- Disassociation from a coordinator: The *MLME-DISASSOCIATE.request* is queued in the indirect queue where it resides until it is polled by the remote device (or the transaction expires).

The IEEE 802.15.4 standard states that a device with a valid short address will supply 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 a short address cannot be matched against the long address. The 802.15.4b task group (see www.ieee802.org/15/pub/TG4b.html) is currently working on a fix but until this change is published and implemented the following limitation exists.

- 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 is possible for all other scenarios. That is, the device may temporarily set its `macShortAddress` to 0xFFFE or 0xFFFF if it wishes to poll for packets queued using the device's extended address.

3.6.1 Disassociation Primitives

This section describes the implementation of the Disassociation related primitives.

3.6.1.1 Disassociate Request

```
// Type: gMlmeDisassociateReq_c
typedef struct mlmeDisassociateReq_tag {
  uint8_t  deviceAddress[8];
  bool_t   securityEnable;
  uint8_t  disassociateReason;
} mlmeDisassociateReq_t;
```

3.6.1.2 Disassociate Indication

```
// Type: gNwkDisassociateInd_c
typedef struct nwkDisassociateInd_tag {
  uint8_t  deviceAddress[8];
  bool_t   securityUse;
  uint8_t  aclEntry;
  uint8_t  disassociateReason;
} nwkDisassociateInd_t;
```

3.6.1.3 Disassociate Confirm

```
// Type: gNwkDisassociateCnf_c
typedef struct nwkDisassociateCnf_tag {
  uint8_t  status;
} nwkDisassociateCnf_t;
```

3.7 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 buffers mainly used for holding Tx or Rx packets and they are limited to a specific number for each device type.
Table 7 shows the number of available large data buffers. The numbers vary because the different device types have more or less functionality.

Table 7. Available Large Data Buffers

Device type	RFD	RFDNB	RFDNBNS	FFD	FFDNGTS	FFDNB	FFDNBNS
Buffers	4	3	3	5	5	4	4

Each time an MCPS-DATA.confirm or MLME-POLL.request primitive is executed one large buffer is used. Even though not directly supported by the IEEE 802.15.4 standard, it is possible to execute an 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 non-beacon mode as device. This means that it is safe for the application to allocate data buffers using the `MSG_AllocType()` function until receiving NULL, indicating that no buffers are available.

3.7.1 Data Primitives

This section describes the implementation of the Data related primitives.

3.7.1.1 Data Request

The Data-Request message structure has an embedded data field. The total size of the message is `(sizeof(mcpsDataReq_t) - 1) + msduLength`. The data field is simply addressed with: `mcpsDataReq->msdu`, and may contain more than one byte even though the array is declared with a size of 1.

```
// Type: gMcpsDataReq_c,
typedef struct mcpsDataReq_tag {
  uint8_t dstAddr[8]; //Address as defined by dstAddrMode
  uint8_t dstPanId[2];
  uint8_t dstAddrMode;
  uint8_t srcAddr[8]; //Address as defined by srcAddrMode
  uint8_t srcPanId[2];
  uint8_t srcAddrMode;
  uint8_t msduLength;  // 0-102
  uint8_t msduHandle;
  uint8_t txOptions;
  uint8_t msdu[1];     // Data will start at this byte
} mcpsDataReq_t;
```

3.7.1.2 Data Confirm

```
// Type: gMcpsDataCnf_c,
typedef struct mcpsDataCnf_tag {
  uint8_t  msduHandle;
  uint8_t  status;
} mcpsDataCnf_t;
```

3.7.1.3 Data Indication

The Data-Indication structure has an embedded data field. The total size of the message is `(sizeof(mcpsDataInd_t) - 1) + (mcpsDataInd->msduLength)`. The data field is simply addressed with: `mcpsDataInd->msdu`, and may contain more than one byte even though the array is declared with a size of 1.

```
// Type: gMcpsDataInd_c,
typedef struct mcpsDataInd_tag {
  uint8_t dstAddr[8]; //Address as defined by dstAddrMode
  uint8_t dstPanId[2];
  uint8_t dstAddrMode;
  uint8_t srcAddr[8]; //Address as defined by srcAddrMode
  uint8_t srcPanId[2];
  uint8_t srcAddrMode;
  uint8_t msduLength;  // 0-102
  uint8_t mpduLinkQuality;
  bool_t  securityUse;
  uint8_t aclEntry;
  uint8_t msdu[1];     // Data will start at this byte
 } mcpsDataInd_t;
```

3.7.1.4 Poll Request

```
// Type: gMlmePollReq_c,
typedef struct mlmePollReq_tag {
  uint8_t  coordAddress[8];
  uint8_t  coordPanId[2];
  uint8_t  coordAddrMode;
  bool_t   securityEnable;
} mlmePollReq_t;
```

3.7.1.5 Poll Confirm

```
// Type: gNwkPollCnf_c,
typedef struct nwkPollCnf_tag {
  uint8_t  status;
} nwkPollCnf_t;
```

3.7.1.6 Communications Status Indication

```
// Type: gNwkCommStatusInd_c,
typedef struct nwkCommStatusInd_tag {
  uint8_t  srcAddress[8];
  uint8_t  panId[2];
  uint8_t  srcAddrMode;
  uint8_t  destAddress[8];
  uint8_t  destAddrMode;
  uint8_t  status;
} nwkCommStatusInd_t;
```

3.7.2 Data Example

The following is an example of the NWK sending an MCPS-DATA.request to the MCPS. It mainly demonstrates how to properly allocate a message buffer, and adding data to the MSDU parameter.

```
// Send 10 bytes of data to MCPS, security is not
// enabled so no need to allocate extra space.
nwkToMcpsMessage_t pMsg =
MSG_Alloc( (sizeof(nwkToMcpsMessage_t)-1) + 10);

// We want to send an MCPS-DATA.request:
pMsg->msgType = gMcpsDataReq_c;

// Fill the MSDU part of the message with some data.
for(i=0; i<10; i++)
  pMsg->msgData.dataReq.msdu[i] = i;

// Set the msdu length
pMsg->msgData.dataReq.msduLength = 10;

// Fill in other fields such as destination+source addresses
CreateDataRequestMessageHeader(pMsg);

// Send message to MCPS. The MCPS will free the message buffer.
MSG_Send(NWK_MCPS, pMsg);
```

The following is an example of the NWK receiving an MCPS-DATA.indication from the MCPS. It shows how the MCPS to NWK SAP handler may be implemented by the application/NWK programmer.

```
// NWK – Receive data indication with 10 bytes of data
uint8_t MCPS_NWK_SapHandler(mcpsToNwkMessage_t *pMsg)
{
  switch(pMsg->msgType) {
  case gMcpsDataInd_c:
    // Handle the incoming data frame
    for(i=0; i<10; i++)
      myBuffer[i] = pMsg->msgData.dataInd.msdu[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;
  }
  MSG_Free(pMsg); // Free message ASAP.
  return gSuccess_c;
}
```

3.8 Purge Feature

The purge feature allows the next higher layer to purge a data packet (MSDU) stored in the MAC until it has been sent. This means that if a MCPS-DATA.request primitive with that

msduHandle has been initiated, it is possible to purge the MSDU with the given msduHandle, if it has not been sent. Initiating the MCPS-PURGE.request primitive and specifying the msduHandle parameter will accomplish 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.

3.8.1 Purge Primitives

This section describes the implementation of the Purge related primitives.

3.8.1.1 Purge Request

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

3.8.1.2 Purge Confirm

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

3.9 Rx Enable Feature

The Rx enable feature allows the network layer to enable the receiver at a given time. The feature is implemented according to the IEEE 802.15.4 standard (section 7.1.10).

3.9.1.1 RX Enable Request

```
// Type: gMlmeRxEnableReq_c,
typedef struct mlmeRxEnableReq_tag {
  bool_t   deferPermit;
  uint8_t  rxOnTime[3];
  uint8_t  rxOnDuration[3];
} mlmeRxEnableReq_t;
```

3.9.1.2 RX Enable Confirm

```
// Type: gNwkRxEnableCnf_c,
typedef struct nwkRxEnableCnf_tag {
  uint8_t  status;
} nwkRxEnableCnf_t;
```

3.10 Guaranteed Time Slots (GTS) Feature

The Guaranteed Time Slots (GTS) feature is a mechanism that allows a device to reserve a certain bandwidth. A GTS slot is always unidirectional and it is always requested by the device.

3.10.1 GTS as 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 the following results.

- The MCPS-DATA.request does not support any method for selecting between Tx slots.
- If the PAN coordinator de-allocates or realigns one of the Tx slots it is not possible to tell which of the slots were affected.

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

Refer to the IEEE 802.15.4 standard, Section 7.5.7 for more information. Allocating a GTS slot or de-allocating 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 very similar. Notice that the local GTS “context” is marked as invalid before the request is actually sent to the PAN coordinator. Any packets that may have been 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 may 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 rather intensive and cannot be completed in IRQ context. The following steps describe the procedure for this software implementation.

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 pre-processed. Pre-processing consists of only one thing: Check if the device's short address is present in the list! An internal flag (`gMlmeGtsAccess`) 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 will asynchronously perform further processing of the beacon frame. This includes generating MLME-BEACON.indications messages to the NWK layer. GTS processing is performed if the `gMlmeGtsAccess` flag was raised. This includes processing all GTS descriptors that matches 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 may have been realigned by the PAN coordinator (that is, a new start slot has been defined). The proper internal GTS context is updated.

3. An existing GTS slot may have been 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 is 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 `gMlmeGtsAccess` flag is cleared.

NOTE

These steps are important because the entire CFP of a superframe will be skipped if the `gMlmeGtsAccess` is detected high at the beginning of the CFP. This is needed because the CFP timing parameters are not yet in place. It is also important that the `Mlme_Main()`is called in a timely manner.

3.10.2 GTS as PAN Coordinator

The PAN coordinator will always accept incoming GTS requests and it will always allocate the requested GTS slot if the minimum length CAP can be maintained. A GTS slot can occupy 1 to15 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. `gMlmeGtsAccess` is raised.
3. The MAC command frame is then queued for further processing by higher layer software (the MLME).
4. This completes GTS processing in IRQ context.

As previously stated, the MLME asynchronously performs further processing of the GTS request. This includes the following 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 a deallocation created “gaps” in the CFP.
3. Timing parameters for the entire CFP is 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.
4. The `gMlmeGtsAccess` flag is cleared.
5. An MLME-GTS.indication message is generated (if applicable).

NOTE

The entire CFP of a superframe will be skipped if the `gMlmeGtsAccess` is detected high at the beginning of the CFP. This is needed because the CFP timing parameters are not yet in place. It is therefore important that the `Mlme_Main()` is called in a timely manner.

Users should also be aware that all existing GTS slots (if any) will be deallocated immediately if the superframe configuration changes (`macBeaconOrder` or `macSuperframeOrder` is changed).

NOTE

Issuing the MLME-START.request primitive updates these two PIB attributes. There is no indication in the beacon frame indicating this. That is, a device must assume that GTS slots have been invalidated if the superframe configuration changes.

GTS expiration is implemented according to the IEEE 802.15.4 standard. The PAN coordinator deallocates stale slots automatically. The IEEE 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.

3.10.3 Miscellaneous Items

The following items are valid for both device and PAN coordinator.

- Although possible to allocate a GTS slot with length 1 at macSuperframeOrder = 0 it is not recommended. This GTS slot will only contain 60 symbols (30 bytes). As already stated, setup time and overhead for PHY and MAC headers are at least 21 bytes, so it will not be possible to send or receive any data. A GTS slot should at least have length = 2 at macSuperframeOrder = 0 (corresponding to 120 symbols). All other superframe orders support GTS slots with length 1.

As previously stated, it is possible so skip a CFP due to GTS maintenance. Although this hazard exists, it should have minimal effects because GTS slots are (probably) rarely allocated and deallocated (mostly at feature setup and termination).

3.10.4 GTS Primitives

3.10.4.1 GTS Request

```
// Type: gMlmeGtsReq_c,
typedef struct mlmeGtsReq_tag {
  bool_t   securityEnable;
  uint8_t  gtsCharacteristics;
} mlmeGtsReq_t;
```

3.10.4.2 GTS Confirm

```
// Type: gNwkGtsCnf_c,
typedef struct nwkGtsCnf_tag {
  uint8_t  status;
```

```
  uint8_t  gtsCharacteristics;
} nwkGtsCnf_t;
```

3.10.4.3 GTS Indication

```
// Type: gNwkGtsInd_c,
typedef struct nwkGtsInd_tag {
  uint8_t  devAddress[2];
  bool_t   securityUse;
  uint8_t  AclEntry;
  uint8_t  gtsCharacteristics;
} nwkGtsInd_t;
```

3.11 Security

The MAC security functionality is implemented as described in the 802.15.4 Standard and in the ZigBee Security Services Specification V.092. Where a different approach is used between the two, the ZigBee Security Services Specification V.092 is followed here. Thus, the CCM* security levels are used in place of the security suites described in the 802.15.4 Standard.
A limitation of the current implementation is that secured beacons will not be processed.
Secured packets are longer when transmitted over the air than corresponding non-secured packets. Besides the increased power consumption and lower maximum throughput, this results in MCPS-DATA.request delivering a `gFrameTooLong_c` error code if the resulting packet gets 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. Table 8 shows the overhead added for each security level.

Table 8. Security Levels

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

3.11.1 Security PIB Attributes

The ACL entries are statically allocated. Currently four ACL entries can be stored in the PIB. It is possible to use the security PIB attributes as defined in **Error! Reference source not found.** with the variation that the ACLSecurityMaterial always takes up the maximum (26 decimal) amount of bytes (DefaultSecurityMaterial does the same, but that is always accessed directly).
It is also possible to write to the contents of the individual ACL entries by use of Freescale specific security PIB attributes, see Section 0.
By setting the Freescale specific `gMacAclEntryCurrent_c` security attribute to an ACL entry index between 0, and 3, it is possible to read and write the individual attributes of the different ACL entries without reading/writing the complete ACL entry descriptor set at once.

Chapter 4
APP/ASP Layer Interface Description

This section describes the Application (APP) / Application Support Package (ASP) interface.

4.1 General APP/ASP Interface Information

The interface between the APP and the ASP is based on service primitives passed from one layer to the other through a layer Service Access Point (SAP). Two SAPs exists, both implemented as functions.

1. `zbErrorCode_t APP_ASP_SapHandler(void *msg)`; APP to ASP SAP `APP_ASP_SapHandler()` passes primitives from the APP to the ASP)

2. `zbErrorCode_t ASP_APP_SapHandler(void *msg)`; ASP to APP SAP `ASP_APP_SapHandler()` passes primitives from the ASP to the APP)

The `APS_APP_SapHandler()` must be implemented in the application layer by the application developer. The SAP handler functions should not be called directly, but through the available macro `MSG_Send(APP_ASP, msg)`. Both APP and ASP service primitives use the same type of messages as defined in the interface header file called AppAspInterface.h. The macros are defined in the header file called PhyMacMsg.h.
Because the APP and ASP interfaces are based on messages being passed to a SAP, each message needs to have an identifier. These identifiers are shown in the two enumerations in Table 9 and Table 10.

Table 9. Primitives in the ASP to APP direction

Message Identifiers	ASP Primitives
gAspAppWakeInd_c	ASP-WAKE.Indication
gAspAppIdleInd_c	ASP-IDLE.Indication
gAspAppInactiveInd_c	ASP-INACTIVE.Indication
gAspAppEventInd_c	ASP-EVENT.Indication

Table 10. Primitives in the APP to ASP direction

Message Identifiers	ASP Primitives
gAppAspGetTimeReq_c	ASP-GETTIME.Request
gAppAspGetInactiveTimeReq_c	ASP-GETINACTIVETIME.Request
gAppAspDozeReq_c	ASP-DOZE.Request
gAppAspAutoDozeReq_c	ASP-AUTODOZE.Request
gAppAspHibernateReq_c	ASP-HIBERNATE.Request
gAppAspWakeReq_c	ASP-WAKE.Request
gAppAspEventReq_c	ASP-EVENT.Request
gAppAspTrimReq_c	ASP-TRIM.Request
gAppAspDdrReq_c	ASP-DDR.Request
gAppAspPortReq_c	ASP-PORT.Request
gAppAspClkoReq_c	ASP-CLKO.Request
gAppAspSetNotifyReq_c	ASP-SETNOTIFY.Request
aspSetMinDozeTimeReq_t	ASP-SET-MIN-DOZE-TIME.Request

The following two sections describe the C-structures which correspond to the message identifiers shown in this section. A common feature of all structures is that all elements of a size greater than 1 byte are little endian and declared as byte arrays.

4.2 ASP to APP Interface

The following structures are used for the messages that go from the ASP to the APP. See the message identifier enumeration list for implemented primitives. All confirm primitives are returned in the same structure as used for the corresponding request. All indication primitives are sent in messages that must be freed by `MSG_Free()`.

4.2.1 Get Time Confirm

This structure contains the result of the ASP-GETTIME.Request. The result is the current value of the MC13192 24 bit event timer (0x000000 to 0xFFFFFF) as a little endian byte array. The status code is always `gSuccess_c`.

```
  // Type: gAspAppGetTimeCfm_c
typedef struct appGetTimeCfm_tag {
  uint8_t status;
  uint8_t time[3];
} appGetTimeCfm_t;
```

4.2.2 Get Inactive Time Confirm

The Get Inactive Time primitive will only yield valid results when called during the inactive portion of a super frame, and MC13192 is not in doze, or hibernate mode. Otherwise, the status parameter will be `gInvalidParameter_c`.

```
  // Type: gAspAppGetInactiveTimeCfm_c
typedef struct appGetInactiveTimeCfm_tag {
  uint8_t status;
  uint8_t time[3];
} appGetInactiveTimeCfm_t;
```

4.2.3 Doze Confirm

The ASP-DOZE.Request primitive specifies the duration in symbols that the MC13192 must doze. If the MC13192 is idle, the requested doze duration is granted and the `actualDozeDuration` parameter is the same as the time requested. However, if the MC13192 has a wait action running with a duration shorter than the requested duration, the MC13192 will doze until the wait action completes. The `actualDozeDuration` parameter will be the difference of the requested doze duration and the end time of the timer action. If doze mode is not possible because the MC13192 is busy, the status parameter is `gInvalidParameter_c`.

```
  // Type: gAspAppDozeCfm_c
typedef struct appDozeCfm_tag {
  uint8_t status;
  uint8_t actualDozeDuration[3];
} appDozeCfm_t;
```

4.2.4 Auto Doze Confirm

The Auto-doze Confirm status is always `gSuccess_c`.

```
  // Type: gAspAppAutoDozeCfm_c
typedef struct appAutoDozeCfm_tag {
  uint8_t status;
} appAutoDozeCfm_t;
```

4.2.5 Hibernate Confirm

The Hibernate Confirm status is gSuccess_c if the MC13192 is in idle mode. Otherwise the return code is `gInvalidParameter_c`.

```
  // Type: gAspAppHibernateCfm_c
typedef struct appHibernateCfm_tag {
  uint8_t status;
} appHibernateCfm_t;
```

4.2.6 Wake Confirm

The Wake Confirm status is always `gSuccess_c`.

```
  // Type: gAspAppWakeCfm_c
typedef struct appWakeCfm_tag {
  uint8_t status;
} appWakeCfm_t;
```

4.2.7 Event Confirm

The Event Confirm message returns the status of the Event request primitive. If the MAC was idle the status parameter is `gSuccess_c`, otherwise it is `gInvalidParameter_c`.

```
  // Type: gAspAppEventCfm_c
typedef struct appEventCfm_tag {
  uint8_t  status;
} appEventCfm_t;
```

4.2.8 Trim Confirm

The Trim Confirm status is always `gSuccess_c`.

```
  // Type: gAspAppTrimCfm_c
typedef struct appTrimCfm_tag {
  uint8_t  status;
} appTrimCfm_t;
```

4.2.9 DDR Confirm

The DDR Confirm status is always `gSuccess_c`.

```
  // Type: gAspAppDdrCfm_c
typedef struct appDdrCfm_tag {
  uint8_t  status;
} appDdrCfm_t;
```

4.2.10 Port Confirm

The Port Confirm status is always `gSuccess_c`.

```
  // Type: gAspAppPortCfm_c
typedef struct appPortCfm_tag {
  uint8_t  status;
  uint8_t  portResult;
} appPortCfm_t;
```

4.2.11 CLKO Confirm

The CLKO Confirm message returns the status of the CLKO request primitive. If an invalid paramter was passed the status parameter is `gInvalidParameter_c`, otherwise it is `gSuccess_c`.

```
  // Type: gAspAppClkoCfm_c
typedef struct appClkoCfm_tag {
  uint8_t  status;
} appClkoCfm_t;
```

4.2.12 Set Notify Confirm

Returns the status of the ASP-SETNOTIFY.Request. If beacons are part of the MAC feature set the status is always `gSuccess_c`. Otherwise the return code is `gInvalidParameter_c`.

```
  // Type: gAspAppSetNotifyCfm_c
typedef struct appSetNotifyCfm_tag {
  uint8_t  status;
} appSetNotifyCfm_t;
```

4.2.13 Set Min Doze Time Confirm

The Set Min Doze Time Confirm status is always `gSuccess_c`.

```
  // Type: gAspAppSetNotifyCfm_c
typedef struct appSetNotifyCfm_tag {
  uint8_t  status;
} appSetNotifyCfm_t;
```

4.2.14 Wake Indication

The ASP-WAKE.Indication primitive is sent to the APP when the MC13192 comes out of doze or hibernate mode. If auto doze is enabled by issuing the ASP-AUTODOZE.Request with the `enableWakeIndication`, and the `autoEnable` parameters set to TRUE, then wake indications are sent to the APP each time auto doze switches from doze to active mode. Auto doze may place the MC13192 in doze mode again after the wake indication has been processed by the ASP_APP SAP. Thus, the APP has the oppertunity to disable auto doze or change the parameters at this time by sending an ASP-AUTODOZE.Request with the new set of parameters. Remember to free this message by calling `MSG_Free()`.

```
  // Type: gAspAppWakeInd_c
typedef struct appWakeInd_tag {
  uint8_t status;
} appWakeInd_t;
```

4.2.15 Idle Indication

This indication is sent to the APP if enabled by the ASP-SETNOTIFY.Request, and the MAC is operating in beaconed mode. The indication is sent at the start of the idle portion of the super frame. The `timeRemaining` parameter is the number of symbols left of the CAP. If macRxOnWhenIdle is TRUE the CAP idle state does not exist, and no idle indications will be sent.
Remember to free this message by calling `MSG_Free()`.

```
  // Type: gAspAppIdleInd_c
typedef struct appIdleInd_tag {
  uint8_t timeRemaining[3];
} appIdleInd_t;
```

4.2.16 Inactive Indication

This indication is sent to the APP if enabled by the ASP-SETNOTIFY.Request, and the MAC is operating in beaconed mode. The indication is sent at the start of the inactive portion of the super frame. The `timeRemaining` parameter is the number of symbols left in the inactive period.
Remember to free this message by calling `MSG_Free()`.

```
  // Type: gAspAppInactiveInd_c
typedef struct appInactiveInd_tag {
  uint8_t timeRemaining[3];
} appInactiveInd_t;
```

4.2.17 Event Indication

This indication is sent to the APP when the requested event has expired.
Remember to free this message by calling `MSG_Free()`.

```
  // Type: gAspAppEventInd_c
typedef struct appEventInd_tag {
  uint8_t dummy; // This primitive has no parameters.
} appEventInd_t;
```

4.2.18 ASP to APP message union

```
// ASP to application message
typedef struct aspToAppMsg_tag {
  uint8_t msgType;
  union {
    appGetTimeCfm_t          appGetTimeCfm;
    appGetInactiveTimeCfm_t appGetInactiveTimeCfm;
    appDozeCfm_t             appDozeCfm;
    appAutoDozeCfm_t         appAutoDozeCfm;
    appHibernateCfm_t        appHibernateCfm;
    appWakeCfm_t             appWakeCfm;
    appEventCfm_t            appEventCfm;
    appTrimCfm_t             appTrimCfm;
    appDdrCfm_t              appDdrCfm;
    appPortCfm_t             appPortCfm;
    appClkoCfm_t             appClkoCfm;
    appSetNotifyCfm_t        appSetNotifyCfm;
    appWakeInd_t             appWakeInd;
    appIdleInd_t             appIdleInd;
    appInactiveInd_t         appInactiveInd;
    appEventInd_t            appEventInd;
  } msgData;
} aspToAppMsg_t;
```

4.2.19 Examples of ASP to APP messages

This section shows examples of how the APP layer should process incoming messages. The examples are not guaranteed to compile because they may contain pseudo code for clarity. Only indications must be handled by the APP SAP. Because ASP requests are performed synchronously, the confirm messages are returned in the message buffer used for the requests, and does thus not end up in the APP SAP.

Example #1: Handle wake indications

```
// APP must have its own SAP handler:
uint8_t ASP_APP_SapHandler(aspToAppMsg_t *pMsg)
{
  // Declared somewhere else
  extern bool_t weAreAutoDozing;

  // Check which indication was received.
  switch(pMsg->msgType) {
  case aspAppWakeInd_t:
    if(weAreAutoDozing == TRUE) {
      // Awoke while auto doze was active. Now do some
 // processing (put data in queue etc.) before MC13192
 // is re-entering doze mode.
      DoSomeThingWhileAwake();
      // When returning from here, MC13192 enters doze mode asap.
      }
    else {
      // MC13192 came out of normal doze or hibernate mode.
    }
    break;
  case aspAppIdleInd_t:
    // ASP-SetNotify.Request(gAspNotifyIdle_c) was issued.
    break;
```

```
  case aspAppInactiveInd_t:
    // ASP-SetNotify.Request(gAspNotifyInactive_c) was issued.
    break;
  case aspAppEventInd_t:
    // ASP-Event.Request(time) was issued.
    break;
  }
  // ALWAYS free incoming messages.
  MSG_Free(pMsg);
  return gSuccess_c;
}
```

4.3 APP to ASP Interface

The following structures are used for the messages that go from the APP to the ASP. See the message identifier enumeration list in Table 10 for implemented primitives. None of the requests in this section are freed by the ASP layer. Thus, the APP layer does not need to use `MSG_Alloc()` for obtaining a buffer for the ASP message. Notice that the ASP layer will modify the message to contain the confirm-primitive corresponding to the request.

4.3.1 Get Time Request

This primitive returns the current value of the MC13192 event timer. See Section 4.2.1 for information about the returned value.

```
// Type: gAppAspGetTimeReq_c
typedef struct aspGetTimeReq_tag {
  uint8_t dummy; // This primitive takes no parameters
} aspGetTimeReq_t;
```

4.3.2 Get Inactive Time Request

This primitive will request the remaining time in the inactive portion of the super frame. See Section 4.2.2 for information about the returned value.

```
// Type: gAppAspGetInactiveTimeReq_c
typedef struct aspGetInactiveTimeReq_tag {
  uint8_t dummy; // This primitive takes no parameters
} aspGetInactiveTimeReq_t;
```

4.3.3 Doze Request

Shut down the MC13192 for a given amount of time in symbols. The CLKO output pin will stop providing a clock signal to the CPU. The `dozeDuration` parameter is the maximum time in number of symbols that the MC13192 will be in doze mode. The MC13192 can be woken up prematurely from doze mode by a signal on the ATTN pin or if a wait action expires in the MAC. CLKO is automatically started again when MC13192 leaves doze mode.

```
// Type: gAppAspDoze_c
typedef struct aspDozeReq_tag {
    // Doze duration in little endian 24 bit symbol time (1symbol=16us)
  uint8_t dozeDuration[3];
} aspDozeReq_t;
```

4.3.4 Auto Doze Request

Automatically shut down the MC13192 during idle periods or if the MAC is in executing a wait action. The CLKO output pin will stop providing a clock signal to the CPU. The `autoDozeInterval` parameter is a suggested period in symbols in which the MC13192 will be in doze mode. This interval may be overridden if doze mode is interrupted by an external signal (ATTN pin) or if a wait action expires in the MAC. If the `enableWakeIndication` parameter is TRUE then an ASP-WAKE.Indication is sent to the APP layer each time the doze interval expires. The indication can be used by the APP layer to do processing. In order to enable auto doze the `autoEnable` parameter must be TRUE. Auto doze can be disabled by sending another ASP-AUTODOZE.Request with the `autoEnable` parameter set to FALSE. It is recommended to use the ASP-WAKE. Indication for simple processing during auto doze since it will occur frequently (if enabled) and the auto doze feature is blocked during the processing of the indication in the ASP_APP SAP.

```
  // Type: gAppAspAutoDoze_c
typedef struct aspAutoDozeReq_tag {
  bool_t autoEnable;
  bool_t  enableWakeIndication;
  uint8_t autoDozeInterval[3];
} aspAutoDozeReq_t;
```

4.3.5 Hibernate Request

The hibernate request shuts down the MC13192. The CLKO output pin will stop providing a clock signal to the CPU. Only a signal on the ATTN pin of the MC13192 or a power loss can bring the MC13192 out of hibernate mode. CLKO is automatically started again when MC13192 leaves hibernate mode. The hibernate mode is not adequate for beaconed operation. Doze mode should be used instead when MC13192 timers are required.

```
  // Type: gAppAspHibernate_c
typedef struct aspHibernateReq_tag {
  uint8_t dummy; // This primitive takes no parameters
} aspHibernateReq_t;
```

4.3.6 Wake Request

Wake-up the MC13192 from Doze/Hibernate mode.

```
  // Type: gAppAspWake_c
typedef struct aspWakeReq_tag {
  uint8_t dummy; // This primitive takes no parameters
} aspWakeReq_t;
```

4.3.7 Event Request

This primitive can be used to schedule a notification for an application event. The primitive is a single instance event. If there is any conflict with the MAC operation a status of `gInvalidParameter_c` is returned, otherwise a status of `gSuccess_c` is returned.
The eventTime parameter is a 3-byte little endian integer symbol time.

```
  // Type: gAppAspEventReq_c
typedef struct aspEventReq_tag {
  uint8_t eventTime[3];
} aspEventReq_t;
```

4.3.8 Trim Request

This primitive sets the trim capacitor value for the MC13192. Upon receipt of the request, the trim capacitor value contained in the 8-bit parameter is programmed to the MC13192.

```
  // Type: gAppAspTrimReq_c
typedef struct aspTrimReq_tag {
  uint8_t trimValue;
} aspTrimReq_t;
```

4.3.9 DDR Request

This primitive sets the GPIO data direction. GPIOs 3-7 are programmed as outputs if the respective bit in mask is a logical 1, otherwise they are programmed as inputs. Bits 0,1 and 2 of mask are ignored.

```
  // Type: gAppAspDdrReq_c
typedef struct aspDdrReq_tag {
  uint8_t directionMask;
} aspDdrReq_t;
```

4.3.10 Port Request

This primitive reads or writes the GPIO data register. portWrite is a Boolean value. If TRUE, the respective bits in `portValue` will be programmed to the GPIO data register (only bits 3-7 are valid).

```
  // Type: gAppAspPortReq_c
typedef struct aspPortReq_tag {
  uint8_t portWrite;
  uint8_t portValue;
} aspPortReq_t;
```

4.3.11 CLKO Request

This primitive sets and/or enables the CLKO output of the MC13192. If `clkoEnable` is TRUE, CLKO is made active, otherwise it is disabled. The CLKO output frequency is programmed depending upon the value contained in `clkoRate` per the CLKO frequency selection of the MC13192.
If an invalid parameter is passed, a status of `gInvalidParameter_c` is returned, otherwise a status of `gSuccess_c` is returned.

```
  // Type: gAppAspClkoReq_c
typedef struct aspClkoReq_tag {
  bool_t  clkoEnable;
  uint8_t clkoRate;
} aspClkoReq_t;
```

4.3.12 Set Notify Request

This primitive controls the indications generated in beaconed operation. The notifications parameter can be any of the following four values:

1. `gAspNotifyNone_c` No indications are sent to the APP layer.
2. `gAspNotifyIdle_c` ASP-IDLE Indication (See Section 4.2.15) is sent.
3. `gAspNotifyInactive_c` ASP-INACTIVE Indication (See Section 4.2.16) is sent.
4. `gAspNotifyIdleInactive_c` ASP-IDLE, and ASP-INACTIVE Indications are sent.

If the MAC PIB attribute macRxOnWhenIdle is set then no idle indications are sent.

```
  // Type: gAppAspSetNotifyReq_c
typedef struct aspSetNotifyReq_tag {
  uint8_t notifications;
} aspSetNotifyReq_t;
```

4.3.13 Set Min Doze Time Request

Set the default minimum doze time. If the allowed doze duration during a MEM WAIT action is smaller than the minimum doze duration, then the MC13192 will NOT be put in to doze mode.

```
  // Type: gAppAspSetMinDozeTimeReq_c
typedef struct aspSetMinDozeTimeReq_tag {
  uint8_t minDozeTime[3]; // Should be > 2ms ((2*1000)/16 = 125)
                          // Default is 4ms.
} aspSetMinDozeTimeReq_t;
```

4.3.14 APP to ASP message union

Use the `aspMsg_t` structure for ASP requests instead of the following `appToAspMsg_t` structure because the request buffer will contain the confirm message after returning from the APP_ASP SAP function call `(MSG_Send(APP_ASP, pMsg) )`. See Section 4.3.15 for examples of sending ASP requests and receiving the corresponding confirm messages.

```
  // APP to ASP message
typedef struct appToAspMsg_tag {
  uint8_t msgType;
  union {
    aspGetTimeReq_t          aspGetTimeReq;
    aspGetInactiveTimeReq_t aspGetInactiveTimeReq;
    aspDozeReq_t             aspDozeReq;
    aspAutoDozeReq_t         aspAutoDozeReq;
    aspHibernateReq_t        aspHibernateReq;
    aspWakeReq_t             aspWakeReq;
    aspEventReq_t            aspEventReq;
    aspTrimReq_t             aspTrimReq;
    aspDdrReq_t              aspDdrReq;
    aspPortReq_t             aspPortReq;
    aspClkoReq_t             aspClkoReq;
    aspSetNotifyReq_t        aspSetNotifyReq;
    aspSetMinDozeTimeReq_t   aspSetMinDozeTimeReq;
  } msgData;
} appToAspMsg_t;
It is recommened to use the following structure for sending requests to
the ASP layer instead of the appToAspMsg_t structure above.
  // APP to ASP, and ASP to APP union.
typedef union aspMsg_tag {
  uint8_t msgType;
  appToAspMsg_t appToAspMsg;
  aspToAppMsg_t aspToAppMsg;
} aspMsg_t;
```

4.3.15 Examples of APP to ASP messages

This section provides examples of how to interact with the ASP layer. The examples are not guaranteed to compile because they may contain pseudo code for clarity.
Example #1: Getting the current MC13192 clock.

```
// APP layer must allocate a message
aspMsg_t aspGetTime;
uint8_t getTimeStat;
uint8_t currentTime[3];

// Set the type.
aspGetTime.msgType = gAppAspGetTimeReq_c;

// Send the request to the ASP SAP. The 'getTimeStat'
// variable can be omitted because the confirm message
// contains a copy of the status.
getTimeStat = MSG_Send(APP_ASP, &aspGetTime);

// Now aspGetTime contains an appGetTimeCfm_t structure.
// aspGetTime.msgType will be gAspAppGetTimeCfm_c.
// aspGetTime.aspToAppMsg.appGetTimeCfm.status is equal
// to getTimeStat;
// Do something with the received time value.
copy(currentTime, aspGetTime.aspToAppMsg.appGetTimeCfm.time);
```

Example #2: Start Auto Doze.

```
// APP layer must allocate a message
aspMsg_t aspAutoDoze;
aspAutoDozeReq_t *pAutoDozeReq;
bool_t weAreAutoDozing = FALSE;

pAutoDozeReq = &aspAutoDoze.appToAspMsg.aspAutoDozeReq;

// Set the type.
aspAutoDoze.msgType = gAppAspAutoDozeReq_c;

// In this example ASP-WAKE.Indications are enabled,
// and doze interval is 5 seconds ((5*1000000)/16 = 125000
// symbols = 0x04C4B4). It is recommended that scans and
// network formation has been carried out before entering
// auto doze if possible.
pAutoDozeReq->autoEnable = TRUE;
pAutoDozeReq->enableWakeIndication = TRUE;
// Values greater than 1 byte must be little endian byte arrays.
pAutoDozeReq->autoDozeInterval[0] = 0xB4;
pAutoDozeReq->autoDozeInterval[1] = 0x4C;
pAutoDozeReq->autoDozeInterval[2] = 0x04;
// Send the request to the ASP SAP.
MSG_Send(APP_ASP, &aspAutoDoze);

// Now aspAutoDoze contains an appAutoDozeCfm_t structure.
// aspAutoDoze.msgType will be gAspAppAutoDozeCfm_c.
if(aspAutoDoze.aspToAppMsg.appAutoDozeCfm.status == gSuccess_c) {
  weAreAutoDozing = TRUE;
}
```

Chapter 5
Parametric Information

The following list shows the main parametric information for the Freescale IEEE 802.15.4 MAC/PHY software.

1. The MCU must run at a minimum clock frequency of 16 MHz to fulfill the 802.15.4 Standard timing requirement for all 802.15.4 Standard features.

2. Within a period of 64µs, the application must disable the MC13192 interrupts for a maximum duration of 10µs.

3. The frequency of the SPI that connects the HCS08 MCU to the MC13192 must be half of the HCS08 clock speed.