802.15.4 MAC PHY Software

Reference Manual

Document Number: 802154MPSRM
Rev. 2.5
04/2010

ARM POWERED®

freescale™
semiconductor

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064, Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

***For Literature Requests Only*:**
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-521-6274 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including “Typicals”, must be validated for each customer application by customer’s technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

Freescale™ and the Freescale logo are trademarks of Freescale Semiconductor, Inc. All other product or service names are the property of their respective owners.

© Freescale Semiconductor, Inc. 2004, 2005, 2006, 2007, 2008, 2009, 2010. All rights reserved.

Contents

About This Book

Audience . . . vii
Organization . . . vii
Revision History . . . vii
Conventions . . . viii
Definitions, Acronyms, and Abbreviations . . . viii
References . . . ix

Chapter 1 IEEE 802.15.4 MAC/PHY Software Overview

1.1 Understanding the 802.15.4 Standard . . . 1-2
1.2 802.15.4 Standard Differences between 2003 and 2006 . . . 1-4
1.3 System Overview . . . 1-5
1.4 802.15.4 MAC/PHY Software Device Types and Libraries . . . 1-6
1.4.1 Code Size versus 802.15.4 Device Type . . . 1-6
1.4.2 PHY Function . . . 1-8
1.4.3 Available Device Types . . . 1-8
1.5 802.15.4 MAC/PHY Parametric Information . . . 1-10
1.6 802.15.4 MAC/PHY Software Build Environment . . . 1-11
1.6.1 Adding User Applications to the Build Environment . . . 1-11
1.7 Freescale 802.15.4 MAC/PHY HCS08 Software Source File Structure . . . 1-12
1.7.1 Used File Extensions . . . 1-12
1.7.2 Source File Structure for HCS08 Based Platforms . . . 1-12
1.8 Configuring the 802.15.4 MAC/PHY HCS08 Software (Users Hardware Platform) . . . 1-13
1.8.1 Redefining the HCS08 Clock Speed . . . 1-13
1.8.2 Changing the Interconnection Between the HCS08 MCU and the MC1319x or MC1320x Transceiver1-14
1.8.3 HCS08 MCU with the MC1319x or MC1320x Transceiver or MC1321x Antenna Control. . . 1-15

Chapter 2 MAC/Network Layer Interface Description

2.1 General MAC/Network Interface Information . . . 2-1
2.2 Data Types . . . 2-3
2.3 Message Buffer Configuration . . . 2-6
2.4 Message System API . . . 2-8
2.4.1 MM_Init . . . 2-8
2.4.2 MSG_Alloc . . . 2-10
2.4.3 MSG_AllocType . . . 2-10
2.4.4 MM_Alloc . . . 2-10
2.4.5 MM_AllocPool . . . 2-11
2.4.6 MSG_Free . . . 2-12
2.4.7 MM_Free . . . 2-12

2.4.8 MSG_Send . . . 2-12
2.4.9 MSG_InitQueue . . . 2-13
2.4.10 MSG_Queue . . . 2-14
2.4.11 MSG_QueueHead . . . 2-14
2.4.12 MSG_DeQueue . . . 2-15
2.4.13 Message Tracking . . . 2-15
2.4.14 MSG_Pending . . . 2-16

Chapter 3
Interfacing to the 802.15.4 MAC Software

3.1 Interface Overview . . . 3-1
3.1.1 MC1310x, MC1320x, and MC1321x Transceiver IRQ Timing Dependency . . . 3-2
3.1.2 MC1322x Transceiver IRQ Timing Dependency . . . 3-3
3.2 Include Files . . . 3-3
3.3 Source Files . . . 3-3
3.4 MAC API . . . 3-4
3.5 MAC Main Task . . . 3-5
3.6 MLME and MCPS Interface . . . 3-6
3.6.1 Resetting . . . 3-6
3.6.2 Accessing PIB Attributes . . . 3-7
3.6.3 MLME Primitives . . . 3-8
3.6.4 MCPS Primitives . . . 3-10
3.7 ASP Interface . . . 3-11

Chapter 4
Feature Descriptions

4.1 Configuration . . . 4-1
4.1.1 PIB Attributes . . . 4-1
4.1.2 Configuration Primitives . . . 4-6
4.1.3 Configuration Examples . . . 4-8
4.2 Scan Feature . . . 4-9
4.2.1 Common Parts . . . 4-9
4.2.2 Energy Detection Scan . . . 4-9
4.2.3 Active and Passive Scan . . . 4-10
4.2.4 Orphan Scan . . . 4-10
4.2.5 Scan Primitives . . . 4-10
4.3 Start Feature . . . 4-14
4.3.1 Start Primitives . . . 4-14
4.4 Sync Feature . . . 4-16
4.4.1 Synchronization Primitives . . . 4-16
4.5 Association Feature . . . 4-17
4.5.1 Association Primitives . . . 4-19
4.5.2 Associate Example . . . 4-22
4.6 Disassociation Feature . . . 4-22

4.6.1 Disassociation Primitives 4-23
4.7 Data Feature 4-25
4.7.1 Data Primitives 4-25
4.7.2 Data Example 4-29
4.8 Purge Feature 4-30
4.8.1 Purge Primitives 4-30
4.9 Rx Enable Feature. 4-31
4.9.1 RX Enable Request 4-31
4.9.2 RX Enable Confirm 4-31
4.10 Guaranteed Time Slots (GTS) Feature 4-31
4.10.1 GTS as a Device 4-31
4.10.2 GTS as PAN Coordinator 4-33
4.10.3 Miscellaneous Items 4-34
4.10.4 GTS Primitives 4-34
4.11 Security 4-35
4.11.1 Security PIB Attributes 4-36
4.11.2 Security Library 4-37
4.11.3 Counter with CBC-MAC (CCM*) 4-37

Chapter 5
APP/ASP Layer Interface Description

5.1 General APP/ASP Interface Information 5-1
5.1.1 uint8_t ASP_APP_SapHandler(aspToAppMsg_t *pMsg) 5-1
5.2 ASP to APP Interface 5-2
5.2.1 Wake Indication 5-2
5.2.2 Idle Indication 5-2
5.2.3 Inactive Indication 5-2
5.2.4 Event Indication 5-3
5.2.5 ASP to APP Message Union. 5-3
5.2.6 Examples of ASP to APP Messages 5-3
5.3 APP to ASP Interface 5-4
5.3.1 Get MAC Time Functions 5-5
5.3.2 uint8_t Asp_GetInactiveTimeReq(zbClock24_t *time) 5-5
5.3.3 uint8_t Asp_DozeReq(zbClock24_t *dozeDuration, uint8_t clko_en) 5-5
5.3.4 uint8_t Asp_AutoDozeReq(bool_t autoEnable, bool_t enableWakeIndication, zbClock24_t *autoDozeInterval, uint8_t clko_en)5-6
5.3.5 uint8_t Asp_AcomaReq(uint8_t clko_en) 5-6
5.3.6 uint8_t Asp_HibernateReq(void) 5-7
5.3.7 uint8_t Asp_EventReq(zbClock24_t *time). 5-7
5.3.8 Device Reference Oscillator Trim Functions 5-7
5.3.9 uint8_t Asp_SetNotifyReq(uint8_t notifications). 5-8
5.3.10 uint8_t Asp_SetMinDozeTimeReq(zbClock24_t *minDozeTime) 5-8
5.3.11 void Asp_TelecTest(uint8_t mode) 5-8
5.3.12 Asp_TelecSetFreq(uint8_t channel) 5-9

5.3.13 Functions for Setting RF TX Power Level 5-9
5.3.14 uint8_t Asp_GetPowerLevel(void). 5-11
5.3.15 void Asp_SetDemodulatorType(bool_t demDCDenable) 5-11
5.3.16 void Asp_EnableComplementaryPAOutput(bool_t enable) 5-12
5.3.17 uint8_t Asp_ConfigureRFCtlSignals(AspRfSignalType_t signalType, AspRfSignalFunction_t function, bool_t gpioOutput, bool_t gpioOutputHigh)5-12
5.3.18 uint8_t Asp_GetMacStateReq(void). 5-13
5.3.19 void Asp_WakeReq(void) 5-14
5.3.20 HCS08 Platform Transceiver GPIO Functions. 5-14
5.3.21 uint8_t Asp_ClkoReq(bool_t clkoEnable, uint8_t clkoRate) 5-15
5.3.22 Examples of APP to ASP calls 5-15

About This Book

This manual describes Freescale's IEEE™ 802.15.4 Standard (2003 and 2006) compliant MAC/PHY software. The Freescale 802.15.4 MAC/PHY software is designed for use with the following families of short range, low power, 2.4 GHz Industrial, Scientific, and Medical (ISM) band transceivers:

- Freescale MC1319x and MC1320x families, designed for use with the HCS08 Family of MCUs.
- Freescale MC1320x, designed for use with the MCS08QE128 MCU.
- Freescale MC1321x , that incorporates a low power 2.4 GHz radio frequency transceiver and an 8-bit microcontroller into a single LGA package.
- Freescale MC1322x Platform-In-Package, that combines a low power 2.4 GHz frequency transceiver and a 32-bit ARM7 microcontroller into a single LGA package.

Throughout this manual, the term transceiver refers to either the MC1319x, MC1320x, or the internal counterpart inside the MC1321x and MC1322x. .

Audience

This document is intended for 802.15.4 MAC application developers.

Organization

This document is organized into five chapters.

Chapter 1	**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	**Interfacing to the 802.15.4 MAC Software** — This chapter describes how to interface an application to the MAC and how to use the MAC interface functions.
Chapter 4	**Feature Descriptions** — The chapter contains descriptions of the Freescale 802.15.4 MAC/PHY software features, focusing on the implementation specific details of the 802.15.4 Standard.
Chapter 5	**APP/ASP Layer Interface Description** — This section describes the Application (APP)/Application Support Package (ASP) interface.

Revision History

The following table summarizes revisions to this document since the previous release (Rev 2.4).

Revision History

Location	Revision
Section 1.2	Added note about Beacon Mode not supported for MC1322x MAC.

Conventions

This document uses the following notational conventions:

- `Courier monospaced type` indicate commands, command parameters, code examples, expressions, datatypes, 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
ASP	Application Support Package
APP	Application
CAP	Contention Access Period
CFP	Contention Free Period
FFD	Full Function Device as specified in the 802.15.4 Standard.
FFDNGTS	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 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 sources were referenced to produce this book:

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
3. 802.15.4 Media Access Controller (MAC) MyWirelessApp User's Guide, Freescale Semiconductor, 2006, 2007.
4. IEEE 802.15.4 Standard - REV b/D6, April 2006.

Chapter 1
IEEE 802.15.4 MAC/PHY Software Overview

This chapter provides an overview of the 802.15.4 Standard (MAC 2003 and MAC 2006) background, and describes Freescale's 802.15.4 MAC/PHY device types and libraries, parametric details, build environment, source file structure and hardware setup.

NOTE

- Users should become familiar with the *IEEE Std 802.15.4™-2003, Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs)* and/or *IEEE Std 802.15.4™-2006, Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs)* as required
- This document will not detail all the differences between the 802.15.4 Standards for 2003 and 2006, except those relevant to the 802.15.4 MAC/PHY software.

The Freescale 802.15.4 MAC/PHY software targets two different platforms:

- The HC(S)08 8-bit MCU family used with the MC1319x, MC1320x and MC1321x
- The ARM7 32-bit MCU used with the MC1322x family

This manual supports the HC(S)08 and ARM7 32-bit platforms and two 802.15.4 Standards (2003 and 2006).

- The MAC 2006 is only available on the ARM7 platform
  - MAC 2006 requires a full function device
  - Beaconing is not presently supported in MAC 2006
- The MAC 2003 is available on both platforms
  - The HC(S)08 supports all features including beaconing and GTS
  - The ARM7 does not support beaconing
- Users should be cognizant that differences in use and services and differences in the standards as deployed in the software are highlighted as necessary throughout the manual.

1.1 Understanding the 802.15.4 Standard

The 802.15.4 Standard was developed for Wireless Personal Area Networks (WPANs). WPANS convey information over short distances among the participants in the network. They enable small, power efficient, inexpensive solutions to be implemented for a wide range of applications and device types. Some key characteristics of an 802.15.4 Standard network are:

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

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

- **Star Network Topology** — In a star network, all communication in the network is either to or from the PAN coordinator. That is, communication between non-PAN coordinator devices is not possible.
- **Peer-to-Peer Network Topology** — In a peer-to-peer network, communication can occur between any two devices in the network as long as they are within range of one another.

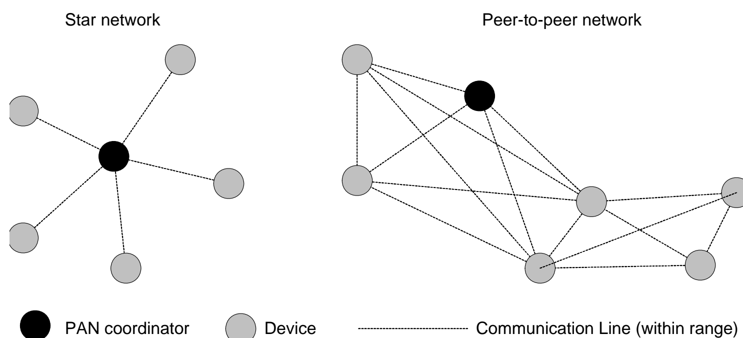

Figure 1-1. Peer-to- Peer and Star Network (No PAN Coordinator)

If a device wants to join an 802.15.4 network it must associate with a device that is already part of the network. In turn, this allows other devices to associate with it. Multiple devices can be associated with the same device as shown in Figure 1-2. A device that has other devices associated with it is a coordinator to those devices. A coordinator can provide synchronization services to the devices that are associated with it through the transmission of beacon frames as shown in Figure 1-2. In a star network there will be only one PAN coordinator, but in a peer-to-peer network there can be multiple coordinators plus the PAN coordinator.

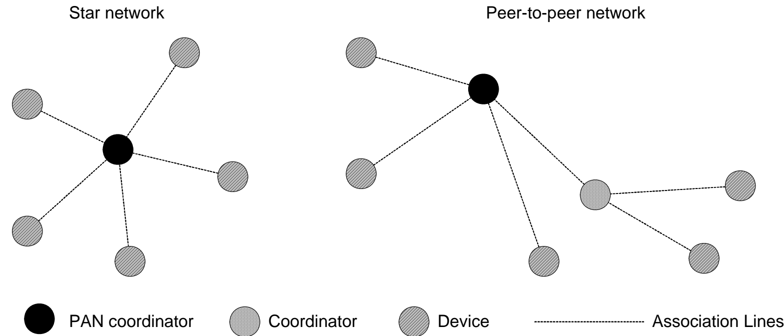

Figure 1-2. Peer and Star Network (With PAN Coordinator)

A network (both star and peer-to-peer) can operate in either beacon mode or non-beacon mode. In beacon mode, all coordinators within the network transmit synchronization frames (beacon frames) to their associated devices and all data transmissions between the coordinator and its associated devices occur in the active period following the beacon frame as shown in Figure 1-3.

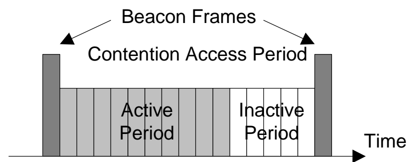

Figure 1-3. Beacon Frame Timing

For both non-beacon and beacon networks, the application can choose to transmit data in the following ways.

- **Direct Data Transfer** — Data from the device to the coordinator using direct data transfer takes place as soon as the channel is free. For beacon networks, direct data is transferred during the active period.
- **Indirect Data Transfer** — Data from a coordinator to a device using indirect data transfer is stored in the coordinator's queue and transferred to the device when the latter does a poll request.
- **Beaconed Tree Mode** — A peer-to-peer network operating in beacon mode will experience beacon collision which can result in the possible loss of synchronization. The ZigBee 1.0 specification outlines the Beaconed Tree Mode, which is a synchronized peer-to-peer network topology. An advantage of a Beaconed Tree Mode network is lower power requirements. A Beaconed Tree Mode network node is active for a short duration (the active portion of the superframe) and it enters a low power mode (sleep) during inactive periods of the superframe. The Freescale 802.15.4 software supports Beaconed Tree Mode as described in the *805.15.4 MyWirelessApp User's Guide* (802154MWASUG).

1.2 802.15.4 Standard Differences between 2003 and 2006

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

- 2006 PHY Enhancements
  - Added a Channel Page to allow more flexibility for new channel allocations
  - Simplified transceiver states (removed Busy_Rx and Busy_Tx)
  - Modified and added PHY PIB attributes
    - Modified phyChannelsSupported attribute
    - Supports PHY PIB access through the MAC SAP
    - Added phyCurrentPage attribute — The current PHY channel page.
    - Added phyMaxFrameDuration attribute — The maximum number of symbols in a frame.
    - Added phySHRDuration attribute — The duration of the sync. header (SHR) in symbols.
    - Added phySymbolsPerOctet attribute — The number of symbols per octet.
- 2006 MAC Enhancements
  - Reduced complexity, reduced MAC overhead and resolved long association times
  - Improved security
  - Supports more detailed beacon scheduling
  - Supports distributed shared (beacon) timebase
  - Supports multicast by employing broadcast frame transmission procedures
  - Provided new CCM suite that consolidates CTR and CBC-MAC suites
  - Removed the Access Control List (ACL)
  - Appended the Auxiliary Security Header (ASH) to the addressing field as part of the MHR
  - Redesigned the MAC security PIB attribute table
  - Clarified security operations and optimized storage of keying material
  - Improved data authenticity and replay protection and simplified protection parameter setup
  - A single key can now be used for different protection levels in a frame
  - Allows unsecured communications until a higher layer sets up the key

NOTE

The MAC used for MC1322x based platforms does not support beacon mode.

1.3 System Overview

Figure 1-4 shows a block diagram of the system. The application uses the lower layers to implement a wireless application based on the Freescale 802.15.4 software.

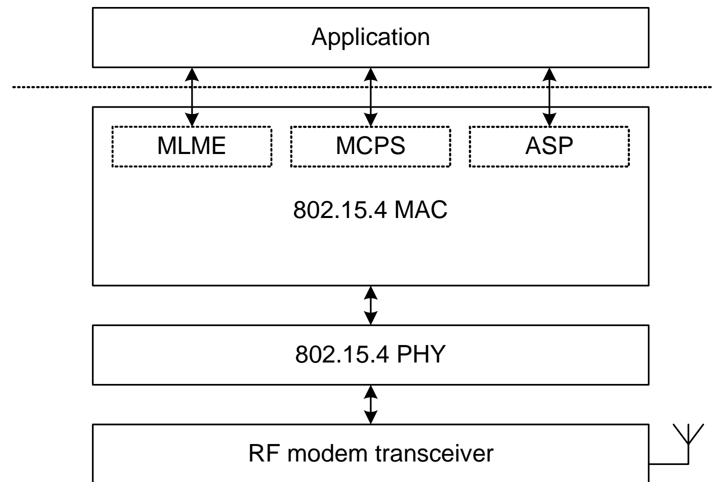

Figure 1-4. System Block Diagram

The application can theoretically be anything and is entirely up to the user. Some examples are:

- Dedicated MAC application
- ZigBee network layer
- Proprietary stack

The layer below the application as shown in Figure 1-4, is the 802.15.4 MAC (or just MAC). The MAC provides three interfaces to the application.

1. **MLME (MAC Sublayer Management Entity) Interface -** This interface is used for all 802.15.4 MAC commands. For example, the application must use this interface to send the MLME-ASSOCIATE.request primitive and it will also receive the MLME-ASSOCIATE.confirm primitive on this interface. This interface is defined in the 802.15.4 Standard.
2. **MCPS (MAC Common Part Sublayer) Interface -** This interface is used for all 802.15.4 data related primitives. The application must use this interface in order to send and receive data. This interface is defined in the 802.15.4 Standard.
3. **ASP (Application Support Package) Interface -** This interface is used for various application support features. For example, the application can request that the hardware enter a low power mode. This interface is proprietary to Freescale.

As shown in Figure 1-4, the two layers at the bottom are the PHY and the actual radio (including hardware driver). The application cannot access the PHY and hardware layer directly.

NOTE

The application must use the three MAC interfaces to implement the desired functionality. Chapter 3, “Interfacing to the 802.15.4 MAC Software” describes the interfaces in complete detail.

1.4 802.15.4 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.

1.4.1 Code Size versus 802.15.4 Device Type

The different 802.15.4 MAC/PHY software Device Types offer various degrees of code sizes by reducing functionality. Table 1-1 (MAC 2003) and Table 1-2 (MAC 2006) shows the relationship between each library and any excluded functionality.

NOTE

- The code sizes for both the HCS08 (MC1319x, 20x, and 21x) and ARM7 (MC1322x) platforms are shown in Table 1-1.
- The code sizes as shown in Table 1-1 are for the complete Freescale 802.15.4 MAC software library. The MAC is available in library format only because it is independent of the hardware platform for the user's 802.15.4 application.
- Table 1-2 shows the ARM7 MC1322x platform FFD library for 2006; 2006 requires full function device and does not presently support beaconing

Table 1-1. MAC/PHY Software Device Type Functionality (MAC 2003)

Device Type	Description	Typical Usage	Mac Library File Name	Code Size	
				MC1319x MC1320x MC1321x	MC1322x[1]
FFD	Full-blown FFD. Contains all 802.15.4 features including security.	PAN Coordinator, Coordinator, Router, or End-device. Includes Beacon Mode support, GTS, parameter verification and security.	802.15.4_MAC_FFD.Lib	37.3 kB	34.3 kB
FFDNB	Same as FFD but no beacon capability.	PAN Coordinator, Coordinator, Router, or End-device. No beacon capability is included, making this Device Type incapable of joining a beacon network. It can transmit/receive beacons for scanning. Includes security and parameter verification.	802.15.4_MAC_FFDNB.Lib	25.9 kB	N/A

Table 1-1. MAC/PHY Software Device Type Functionality (MAC 2003) (continued)

Device Type	Description	Typical Usage	Mac Library File Name	Code Size	
				MC1319x MC1320x MC1321x	MC1322x[1]
FFDNBNS	Same as FFD but no beacon and no security capability.	PAN Coordinator, Coordinator, Router, or End-device. No beacon capability is included, making this Device Type incapable of joining a beacon network. It can transmit/receive beacons for scanning. Security is not supported.	802.15.4_MAC_FFDNBNS.Lib	21.3 kB	N/A
FFDNGTS	Same as FFD but no GTS capability.	PAN Coordinator, Coordinator, Router, or End-device. Lacks the ability to communicate using GTS, but may participate in a Beacon Network. Includes security.	802.15.4_MAC_FFDNGTS.Lib	33.3 kB	N/A
RFD	Reduced function device. Contains 802.15.4 RFD features.	Operates as an End-device only and can participate in beacon networks. Includes security.	802.15.4_MAC_RFD.Lib	27.8kB	N/A
RFDNB	Same as RFD but no beacon capability.	Operates as an End-device only, and can not participate in beacon networks. Includes security	802.15.4_MAC_RFDNB.Lib	23.0kB	N/A
RFDNBNS	Same as RFD but no beacon and no security capability.	Can operate as an End-device only, and can not participate in beacon networks. Security is not supported.	802.15.4_MAC_RFDNBNS.Lib	18.4kB	N/A

[1] MAC 2003 on the ARM7 does not support beaconing/GTS

Table 1-2. MAC/PHY Software Device Type Functionality (MAC 2006)

Device Type	Description	Typical Usage	Mac Library File Name	Code Size	
				MC1319x MC1320x MC1321x	MC1322x[1]
FFD	Full-blown FFD. Contains all 802.15.4 features including security.	PAN Coordinator, Coordinator, Router, or End-device. Includes parameter verification and security.	802.15.4_MAC_FFD.Lib	Not available	46.6 kB

[1] MAC 2006 on the ARM7 does not support beaconing/GTS

1.4.2 PHY Function

The PHY is independent of whether the end user application is a Full Function Device or a Reduced Function Device as well as the platform.

1.4.2.1 HCS08-Based Platforms

The HCS08 PHY is available in source code format because it is dependent on the hardware platform used for the 802.15.4 application. If users want to run the Freescale 802.15.4 MAC/PHY software on their own hardware platform where the MCU to transceiver connections can vary, they may need to change the definition of the connections between the HCS08 MCU and the MC1319x or MC1320x. See Section 1.8, "Configuring the 802.15.4 MAC/PHY HCS08 Software (Users Hardware Platform)" for a detailed description of how to make the Freescale 802.15.4 MAC/PHY software run on the user's own hardware platform.

For the MC1321x platform, the HCS08 MCU and transceiver are internally interfaced with fixed connections in the SiP. However, users may still control items such as antenna control by customizing the PHY driver. See Section 1.8, "Configuring the 802.15.4 MAC/PHY HCS08 Software (Users Hardware Platform)" for a detailed description of how to make modifications to the Freescale 802.15.4 PHY driver software.

1.4.2.2 ARM7 MC1322x Platform

The ARM7 PHY is dependent on the 802.15.4 MAC version:

- The MAC 2003 version is implemented as part of the MAC library in the MC1322x ROM.
- The MAC 2006 version PHY is available only as part of MAC RAM library. It is not available as source. It is exercised directly through the MAC libraries.

1.4.3 Available Device Types

The following sections describe the available device types.

1.4.3.1 Full Function Device (FFD) Device Type

The Freescale 802.15.4 MAC/PHY software FFD type is an 802.15.4 Standard compliant Full Functional Device that includes all MAC features. 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 library to create a device with FFD capabilities.

For ARM7 MC1322x platform, the Full Function Device (FFD) type is the only option when using the 802.15.4 MAC/PHY software. For this platform, the FFD MAC 2003 library is resident in ROM. The MAC 2006 version is available only as a RAM build library.

NOTE

The MAC2006 option is available only as an FFD.

1.4.3.2 Full Function Device With No GTS (FFDxxNGTSxx) Device Types

The Freescale 802.15.4 MAC/PHY software FFDxxNGTSxx Device Types are 802.15.4 Standard compliant FFDs that exclude GTS functionality. These can be used in applications that require both device and coordinator functionality such as ZigBee routers. These libraries cannot be used for applications that require GTS data transmissions.

Users should compile their application with the 802.15.4_MAC_FFDxxNGTSxx libraries to create a device with FFDNGTS capabilities.

1.4.3.3 Full Function Device No Beacon (FFDxxNBxx) Device Types

The Freescale 802.15.4 MAC/PHY software FFDxxNBxx Device Types are 802.15.4 compliant Full Functional Devices that exclude beacon functionality. These can be used in applications that require both device and coordinator functionality such as ZigBee routers. These libraries cannot be used for creating beaconed networks.

Users should compile their application with the 802.15.4_MAC_FFDxxNBxx libraries to create a device with FFDNB capabilities.

1.4.3.4 Full Function Device No Security (FFDxxNSxx) Device Types

The Freescale 802.15.4 MAC/PHY software FFDxxNSxx Device Types are 802.15.4 compliant Full Functional Devices that exclude security functionality. These can be used in applications that require both device and coordinator functionality such as ZigBee routers. These libraries cannot be used for applications that require encrypted or otherwise secured transactions.

Users should link their application with the 802.15.4_MAC_FFDxxNSxx libraries to create a device with FFDNS capabilities.

1.4.3.5 Full Function Device Pan Only (FFDPxx) Device Types

The Freescale 802.15.4 MAC/PHY software FFDPxx Device Types are 802.15.4 compliant Full Functional Devices which can be used in applications that require only the coordinator functionality. These libraries cannot be used for applications that require end device capabilities.

Users should link their application with the 802.15.4_MAC_FFDPxx libraries to create a device with FFDP capabilities.

1.4.3.6 Full Function Device ZigBee Security(FFDxxZSxx) Device Types

The Freescale 802.15.4 MAC/PHY software FFDxxZSxx Device Types are 802.15.4 compliant Full Functional Devices. These can be used in applications that require implementing their own security layer, using the security module provided by the MAC.

Users should link their application with the 802.15.4_MAC_FFDxxZSxx libraries to create a device with FFDZS capabilities.

1.4.3.7 Reduced Function Device (RFD) Device Type

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

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

1.4.3.8 Reduced Function Device No Beacon (RFDxxNBxx) Device Types

The Freescale 802.15.4 MAC/PHY software RFDNB Device Types are 802.15.4 compliant Reduced Functional Device that exclude beacon functionality. These can be used in applications that only require device functionality such as leaf devices (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_RFDxxNBxx libraries to create a device with RFDNB capabilities.

1.4.3.9 Reduced Function Device No Security (RFDxxNSxx) Device Types

The Freescale 802.15.4 MAC/PHY software RFDxxNSxx Device Types are 802.15.4 compliant Reduced Functional Devices that exclude security functionality. These can be used in applications that only require device functionality such as leaf devices (end-devices with no child devices). This library cannot be used for applications that require encrypted or otherwise secured transactions.

Users should compile their application with the 802.15.4_MAC_RFDxxNSxx libraries to create a device with RFDNS capabilities.

1.4.3.10 Reduced Function Device ZigBee Security (RFDxxZSxx) Device Types

The Freescale 802.15.4 MAC/PHY software RFDxxNVxx Device Types are 802.15.4 compliant Reduced Functional Device. These can be used in applications that only require device functionality, such as leaf devices (end-devices with no child devices) and that require implementing their own security layer by using the security module provided in the MAC.

Users should compile their application with the 802.15.4_MAC_RFDxxZSxx libraries to create a device with RFDZS capabilities.

1.5 802.15.4 MAC/PHY Parametric Information

The following lists show the main parametric information for the Freescale 802.15.4 MAC/PHY software.

The clock requirements stated here are for an HC(S)08 based MCU. The HCS08 CPU clock is always 2 times the bus clock. The bus clock is referenced in the following list.

- When running in beacon mode the MCU bus clock must run at a minimum clock frequency of 16 MHz to meet the 802.15.4 Standard timing requirement for all 802.15.4 Standard features
- When running in non-beacon mode the MCU bus clock can also run at a frequency of 8 MHz

- Within a period of 64µs, the application must disable the MC1319x/MC1320x/MC1321x interrupts for a maximum duration of 10µs, when running at 16 MHz bus clock
- Within a period of 64µs, the application must disable the MC1319x/MC1320x/MC1321x interrupts for a maximum duration of 7µs, when running at 8 MHz bus clock
- The frequency of the SPI that connects the HCS08 MCU to the transceiver must be half of the HCS08 bus clock speed
- The maximum allowed time for each Application Task is 4ms. No Application Task should have higher priority than the MAC Task.

  The clock requirements stated here are for an ARM7 based MC1322x platform. The CPU clock, bus clock, and peripheral clock on the MC1322x are always at the same frequency, derived from the reference oscillator. The maximum frequency is the reference oscillator which is typically 24 MHz.
- To ensure that timing constraints are met as required by the 802.15.4 Standard, it is recommended the MCU clocks run at 24MHz.
- The maximum allowed time for each Application Task is 4ms. No Application Task should have higher priority than the MAC Task.
- If MAC 2006 is used, the maximum allowed time for the interrupts to be disabled is 40 µs.

1.6 802.15.4 MAC/PHY Software Build Environment

Freescale 802.15.4 MAC/PHY applications can be generated using the Freescale BeeKit Wireless Connectivity Toolkit. For more information on how to create wireless applications using BeeKit, see the *BeeKit Wireless Connectivity Toolkit User's Guide* (BKWCTKUG).

This section describes the Freescale 802.15.4 MAC/PHY software build environment.

- For HCS08 based platforms, the Freescale 802.15.4 MAC/PHY software is built using the IDE CodeWarrior Development Studio for Freescale HC08. Users should employ the `Freescale_802.15.4_MAC_PHY_V50.mcp` file if development is based on the IDE CodeWarrior Development Studio for Freescale HC08 for MAC 2003 version or `Freescale_802.15.4_MAC_PHY_2006_V60.mcp` for MAC 2006 version.
- For the ARM7 based MC1322x platform, the Freescale 802.15.4 MAC/PHY software is built using the IAR Embedded Workbench IDE.

1.6.1 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 HCS08 802.15.4 PHY source code, and CodeWarrior 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 knowledge of the 802.15.4 Standard 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 MyWirelessApp Software User's Guide* (802154MWASUG). Users can generate MyWirelessApp source code, via Freescale BeeKit.

- For more information, please refer to Freescale ZigBee home page at www.freescale.com/zigbee.

1.7 Freescale 802.15.4 MAC/PHY HCS08 Software Source File Structure

This section describes the source file structure of the Freescale 802.15.4 MAC/PHY software.

1.7.1 Used File Extensions

The Freescale 802.15.4 MAC/PHY software uses the following file extensions:

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

1.7.2 Source File Structure for HCS08 Based Platforms

This section describes the source file structure for applications based on MC1319x, MC1320x and MC1321x platforms.

NOTE

All targets are drive and main directory independent. The `.mcp` project file and the MAC/PHY libraries have a version number added to the end of the file name for version tracking.

The Freescale 802.15.4 MAC/PHY software for HCS08 is arranged in a the following file structure:

```
|——Application
| |——Configure Configuration header files
| |——Init Application initialization code
| |——Interface 802_15_4.h header file
| |——Source Application source files
| |——UartUtil UART helper functions
|——Bin Empty output directory
|——MacStandalone
| |——Interface MAC interface header files
| |——Mac
| | |——802.15.4_MAC_FFD.Lib library
| | |——802.15.4_MAC_FFDNB.Lib library
| | |——802.15.4_MAC_FFDNBNS.Lib library
| | |——802.15.4_MAC_FFDNGTS.Lib library
| | |——802.15.4_MAC_RFD.Lib library
| | |——802.15.4_MAC_RFDNB.Lib library
| | |——802.15.4_MAC_RFDNBNS.Lib library
| |——Phy
| | |——Interface PHY interface header files
```

```
| | |——Isr Interrupt handlers
| | |——Primitives Source code of PHY primitives
|——PLM
| |——Interface Platform interface files
| |——PRM Beestack.prm
| |——Source Source code for drivers
|——SSM
| |——TS Task scheduler source code
| |——ZTC ZigBee test client source code
|——mcp CodeWarrior project file
```

1.8 Configuring the 802.15.4 MAC/PHY HCS08 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 MC1319x or MC1320x. This enables users to run their 802.15.4 application on their own hardware platform.

1.8.1 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, the system clock is controlled from the files in the `sys` directory. Freescale recommends copying and reusing the files from these examples.

The “Type_XXXX” define selects a minimum system bus frequency in the `AppToPlatformConfig.h` header file in the interface directory for each Device Type. The application can choose to use a higher system bus frequency, but Freescale recommends to not use a lower one. If users want to use a higher frequency than necessary, 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

- Some MCU UART serial port baud rates may not be available at certain system clock frequencies. The resulting frequency from the baud rate generator may be too inaccurate to use at a given standard baud rate.
- When using the HCS08 clock generator FLL to generate different clock frequencies, the user must be aware that certain frequencies may violate possible errata conditions for a given HCS08 device. Be sure to check the errata for a given device before choosing a custom clock frequency.

- The 16 MHz default HCS08 bus clock for the MAC, as well as, the above referenced 8 MHz bus clock do not violate any errata conditions for the MCU.

1.8.2 Changing the Interconnection Between the HCS08 MCU and the MC1319x or MC1320x Transceiver

The PHY is provided with standard interconnections between the MCU and the transceiver for a number of Freescale development modules. If users require a different interconnection for their own hardware platform, the PHY files must be changed. For this reason the PHY library is delivered as source code.

NOTE

The MC1321x SiP contains fixed internal interconnection the internal HCS08 MCU and transceiver. Therefore, no changes can be made for this mapping. The NCB and SRB use the MC1321x, so this section is not applicable to those boards.

In the `MacPhy.h` header file, a set of macros is defined which are used directly by the PHY layer for antenna control and other functions. 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 configured in the `PortConfig.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 the `PortConfig.h` file can be changed for a new hardware configuration. All macros use the following definitions 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 mapping
  #define gMC1319xAttnPort      PTBD
  #define gMC1319xRxTxPort      PTBD
  #define gMC1319xResetPort     PTBD

  #define gMC1319xAttnMask_c    (1<<2)
  #define gMC1319xRxTxMask_c    (1<<3)
  #define gMC1319xResetMask_c   (1<<1)

  #define gMC1319xGPIO1Port     PTBD
  #define gMC1319xGPIO2Port     PTBD
  #define gMC1319xAntSwPort     PTBD

  #define gMC1319xGPIO1Mask_c   (1<<4)
  #define gMC1319xGPIO2Mask_c   (1<<5)
  #define gMC1319xAntSwMask_c   (1<<6)

  #define gMC1319xSpiTxD1Mask_c  (1<<0)
  #define gMC1319xSpiRxD1Mask_c  (1<<1)
  #define gMC1319xSpiSsMask_c    (1<<2)
  #define gMC1319xSpiMisoMask_c  (1<<3)
  #define gMC1319xSpiMosiMask_c  (1<<4)
  #define gMC1319xSpiSpsckMask_c (1<<5)
```

1.8.3 HCS08 MCU with the MC1319x or MC1320x Transceiver or MC1321x Antenna Control

The type of antenna (single or dual antenna) is specified by the use of the Dual Antenna field of the gHardwareParametersInit structure, defined in `NV_Data.h`. If the value is FALSE (0), the MAC will use a single antenna and if it is TRUE (1), the MAC will use separate antennas for Tx and Rx.The antenna type can also be specified in BeeKit by setting the “User defined internal or external antenna switch configuration” in the Platform component.

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. uint8_t MLME_NWK_SapHandler(nwkMessage_t *pMsg); MLME to NWK SAP MLME_NWK_SapHandler() function passes primitives from the MLME to the NWK)
2. uint8_t MLME_NWK_SapHandler(nwkMessage_t *pMsg); 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 MSG_Send(SAP msg) macro. The identifier ‘SAP’ will be concatenated with _SapHandler, so the 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 `NwkMacInterface.h` interface header file. The macros are defined in the `MsgSystem.h` header file.

The NWK_MLME_SapHandler() and NWK_MCPS_SapHandler() functions may place a message in a queue. In order to process queued messages, the MAC task needs to run.

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 the application.

Because the NWK and MLME/MCPS interfaces are based on messages being passed to a few SAPs, 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 2-1 lists all the message identifiers in the MLME to NWK direction. They cover all the MLME confirm and indication primitives.

Table 2-1. 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-ORPHAN.Indication
gNwkResetCnf_c	MLME-RESET.Confirm
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
gNwkPollNotifyIndication_c	Freescale proprietary Poll Notify Indication

Table 2-2 lists all the message identifiers in the MCPS to NWK direction. They cover all the MCPS confirm and indication primitives.

Table 2-2. 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.Indication
gMcpsPurgeCnf_c	MCPS-PURGE.Confirm

Table 2-3 lists all the message identifiers in the NWK to the MLME direction. They cover all the MLME request and response primitives.

Table 2-3. 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

Table 2-3. Primitives in the NWK to MLME Direction (continued)

Message Identifier (primNwkToMlme_t)	802.15.4 NWK to MLME Primitives
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 2-4 provides a list of all the message identifiers in the NWK to the MCPS direction. They cover all the MCPS request and response primitives.

Table 2-4. 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, with the exception of the pointer type, 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 pointer type is the exception from the little endian notation. The pointer type 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 the values 0 (No), 2 (Short), and 3 (Extended).

The structures described in Section 4.1.2.1, “Reset Request” through Section 4.10.4.3, “GTS Indication” 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;
    nwkErrorCnf_t           errorCnf;       // Test framework primitive.
    nwkBeaconStartInd_t     beaconStartInd;
    nwkMaintenanceScanCnf_t maintenanceScanCnf;
    nwkPollNotifyInd_t      pollNotifyInd;
  } msgData;
} nwkMessage_t;
```

For messages from the MCPS to the NWK for the S08 platform, 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;
```

For messages from the MCPS to the NWK for the ARM7 platform, 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;
    mcpsPromInd_t   promInd;
    void            *dummyAlign;// Used for aligning union, so that mcpsToNwkMessage_t may be
cast to nwkMessage_t (in PassMacMessageUp of PTC)
  } 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 4.1.2, "Configuration Primitives".

```
// 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 (S08 platform) 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;
```

An allocated message (ARM7 platform) 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;
    void *                dummyAlign; // Used for alignment with mlmeGenericMsg_t
  } msgData;
} nwkToMcpsMessage_t;
```

2.3 Message Buffer Configuration

The message system, which is an integral part of the MAC and the interface to the MAC, relies on a pool of message buffers. Depending on the Device Type and selected feature set the buffer pool varies in size. Typically, a coordinator can have 5 small messages of 22 bytes each and 5 large messages of 134 bytes each. The number of buffers is defined in the AppToMacPhyConfig.h header file in the Configure folder. Specifically, the gTotalSmallMsgs_d, and gTotalBigMsgs_d constant definitions can be used to configure the number of message buffers.

NOTE

Freescale strongly recommends to only increase the number of message buffers because MAC functionality may be adversely affected by a reduction in the number of message buffers. If users are absolutely required to reduce the number of message buffers, then extensive testing must be performed to ensure that the MAC is not influenced by the reduced buffer count.

When building applications incorporating the Freescale MAC, the `GlobalVars.c` source file must be included because it contains the instantiation of the MAC message pools.

An application is allowed to add message buffers to the pools as well as use the extra buffers for both MAC messages and non-MAC related memory allocations. However, if using the buffers for private allocations, the application must take care to not allocate more buffers than was added. For example, if gTotalSmallMsgs_d is changed from 5 to 12, then the application should only use the extra 7 buffers for private allocations. Otherwise, the MAC may fail to function properly.

Instead of adding application specific buffers to the MAC buffer pool, the application can create its own private pool. The following example shows how to accomplish this using the data types and macros from the `MsgSystem.h` file. The example defines four pools configured as shown in Table 2-5.

Table 2-5. Pool Configuration

Pool ID	Number of buffers	Size of each buffer in bytes
0	mMyAppNumBuf0 = 8	mMyAppBufSize0 = 16
1	mMyAppNumBuf1 = 4	mMyAppBufSize1 = 32
2	mMyAppNumBuf2 = 2	mMyAppBufSize2 = 50
3	mMyAppNumBuf3 = 1	mMyAppBufSize3 = 128

The message system header file is included and then the APP_Alloc*** macros are defined to access the application specific pool (myAppPools).

```
#include "MsgSystem.h"

#define APP_Alloc(size) MM_AllocPool(myAppPools, numBytes)
#define APP_AllocType(type) MM_AllocPool(myAppPools, sizeof(type))
```

The pool layout (S08 platform) is defined using the poolInfo_t type shown below:

```
typedef struct poolInfo_tag {
  uint8_t poolSize;
```

```
  uint8_t blockSize;
  uint8_t nextBlockSize;
} poolInfo_t;
```

The pool layout (ARM7 platform) is defined using the poolInfo_t type shown below:

```
typedef struct poolInfo_tag {
  uint8_t poolSize;
  uint8_t blockSize;
  uint8_t nextBlockSize;
  uint8_t padding[1];
} poolInfo_t;
```

NOTE

It is important that the buffer sizes are sorted in ascending order. The sequence in this example must be as follows:
16, 32, 50, 128

```
#define mMyAppNumBuf0   8   // Pool0 has 8 buffers of
#define mMyAppBufSize0 16   // 16 bytes each
#define mMyAppNumBuf1   4   // Pool1 has 4 buffers of
#define mMyAppBufSize1 32   // 32 bytes each
#define mMyAppNumBuf2   2   // Pool2 has 2 buffers of
#define mMyAppBufSize2 50   // 50 bytes each
#define mMyAppNumBuf3   1   // Pool3 has 1 buffer of
#define mMyAppBufSize3 128 // 128 bytes

const poolInfo_t myAppPoolInfo[4] = {
  mMyAppNumBuf0, mMyAppBufSize0, mMyAppBufSize1,
  mMyAppNumBuf1, mMyAppBufSize1, mMyAppBufSize2,
  mMyAppNumBuf2, mMyAppBufSize2, mMyAppBufSize3,
  mMyAppNumBuf3, mMyAppBufSize3, 0
};
```

In order to allocate the heap for the pools, the total size of the pools must be known. The following code snippet shows how to calculate the total heap size and then allocate it. Extra space (sizeof listHeader_t) is added to each buffer for storing two pointers related to linked list operations.

```
#define mMyAppHeapSize (\
  mMyAppNumBuf0*(mMyAppBufSize0+sizeof(listHeader_t)) + \
  mMyAppNumBuf1*(mMyAppBufSize1+sizeof(listHeader_t)) + \
  mMyAppNumBuf2*(mMyAppBufSize2+sizeof(listHeader_t)) + \
  mMyAppNumBuf3*(mMyAppBufSize3+sizeof(listHeader_t)) )

uint8_t myAppHeap[mMyAppHeapSize];
```

NOTE

Depending on machine architecture, it may be required to restrict the buffer sizes to values divisible by sizeof(void *). Otherwise, bus access violations may occur.
For example, a buffer size of 50 can cause problems on 32 bit architectures. To avoid potential issues, the size should be defined as 32.

When initializing the pools the myAppPools array is filled with information about each pool. This array is used as the handle to the application's private pool when allocating a buffer.

```
pools_t myAppPools[4];
```

```
  // Initialize application pools
MM_Init(myAppHeap, myAppPoolInfo, myAppPools);
```

When allocating buffers from the application pool, use the APP_Alloc(size) and APP_AllocType(type) macros. They will translate to MM_Alloc(myAppPools, size), and MM_Alloc(myAppPools, sizeof(type)) respectively. MSG_Free can be used on all buffers regardless of the pool they originate from. This implies that the application private buffers can be sent to the MAC as long as the message sizes expected by the MAC are complied with (a minimum of 22 bytes for small non-data messages and a minimum of 134 bytes for data packets and command frames). When freed by the MAC, the application buffers are returned to the applications private pool.

2.4 Message System API

This section describes the macros and functions available in the Message System API. In order to use the API, the `MsgSystem.h` header file must be included in the relevant source code files. The `GlobalVars.c` file is also a requirement in the build.

2.4.1 MM_Init

Prototype

```
void MM_Init(uint8_t *pHeap, const poolInfo_t *pPoolInfo, pools_t *pPools);
```

pHeap

Points to a contiguous memory block with space for the complete memory pool including space for linked list housekeeping. The number of bytes required in the heap can be calculated using the following equation:

$$HeapSize = \sum_{m=0}^{m=M-1} N_m (S_m + sizeof(listHeader_t))$$

Where:

M is the number of pools of different buffer sizes.

N_m is the number of buffers in pool m.

S_m is the number of bytes in each buffer in pool m.

For example, if M=2, pools are defined with N_0=3 buffers of S_0=16 bytes, and N_1=2 buffers of S_1=64 bytes, then the amount of heap memory must be: 3*(16+4) + 2*(64+4) = 196 bytes, assuming that sizeof(listHeader_t) is 4. The heap can be created as follows:

```
uint8_t myHeap[196];
```

pPoolInfo

Points to an array of data structures which define the memory pool layout. The array is not altered by this function and may be placed in read only memory. The format of the pool info array is shown in Table 2-6.

Table 2-6. Pool Info Array

N_0	S_0	S_1
N_1	S_1	S_2
N_2	S_2	S_{M-1}
N_{M-1}	S_{M-1}	0 (Termination)

An important restriction to the buffer sizes is shown in the following formula:

$$S_m \le S_{m+1}$$

For example, the pools must be defined with ascending buffer sizes. An equally important restriction is that the buffer size must be a modulus of the pointer size of the machine architecture. For example, on 32 bit MCUs the buffer size must be divisible by 4. Otherwise, bus access violations occur when accessing misaligned buffers.

Using the previous example, a pool can be constructed by defining the following structure:

```
const poolInfo_t myPoolInfo[2] =     {
 3, 16, 64,
 2, 64, 0                            };
```

pPools

Points to an array of c-structures which will receive the initialized memory pool handle. The handle is used when allocating memory from the pool. If M pools have been defined in the pool info array, then the output array also needs to reserve space for M number of pools_t structures. For example, continuing with the current example, the array is defined as:

```
pools_t myPools[2];
```

To fill in the array the MM_Init function is called as follows:

```
MM_Init(myHeap, myPoolInfo, myPools);
```

Now allocation and deallocation is possible using the application specific allocation function and the standard free function:

```
myDataType_t *pBuffer = MM_AllocPool(myPools, 12);
MSG_Free(pBuffer);
```

Functional Description

The MM_Init function is used whenever a new set of memory pools must be created. It is used by the MAC during initialization and soft reset (MLME-RESET.request) to configure the MAC message pool. However, applications may also use the function for creating their own private pools.

The array of poolInfo_t structures is used for segmenting the supplied heap into buffers, and organizing them in pools. All buffers are assigned a header with a pointer to the original pool, and a next pointer for linked list operations. The pool information, including anchors for each buffer pool, is stored in the output pools_t array.

2.4.2 MSG_Alloc

Macro Definition

```
#define MSG_Alloc(size) MM_Alloc(size)
```

Size

Specifies the size of the requested buffer, which must be less than 256 bytes.

Functional Description

The macro is typically translated directly to MM_Alloc(size). See Section 2.4.4, "MM_Alloc" for more information.

Example

```
uint8_t *pMsg = MSG_Alloc(10);
```

2.4.3 MSG_AllocType

Macro Definition

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

Type

Specifies the type of the requested buffer. The type is translated to a size which must be less than 256 bytes.

Functional Description

The macro is typically translated directly to MM_Alloc(sizeof(type)). See Section 2.4.5, "MM_AllocPool" for more information.

Example

```
mlmeMessage_t *pMsg = MSG_AllocType(mlmeMessage_t);
```

2.4.4 MM_Alloc

Prototype

```
void *MM_Alloc(uint8_t size);
```

Size

Specifies the size of the requested buffer, which must be less than 256 bytes.

Returns

Pointer to allocated buffer, or NULL if no buffers were available.

Functional Description

This function is solely for allocating buffers from the MAC pools. Otherwise, the functionality is identical to MM_AllocPool. See Section 2.4.5, "MM_AllocPool" for more information. Freescale recommends using the MSG_Alloc and MSG_AllocType macros instead of the direct function call.

Example

```
nwkToMcpsMessage_t *pMsg = MM_Alloc(sizeof(nwkToMcpsMessage_t));
```

2.4.5 MM_AllocPool

Prototype

```
void *MM_AllocPool(pools_t *pPool, uint8_t size);
```

pPool

The pool to allocate a buffer from. pPool must have been initialized by MM_Init.

Size

Specifies the size of the requested buffer, which must be less than 256 bytes.

Returns

Pointer to allocated buffer, or NULL if no buffers were available.

Functional Description

This function searches in the memory pool specified by the pPool argument for a buffer of at least the number of bytes given by the size argument. The search starts at the smallest buffer size and ends at the largest buffer size.

For example, assume a memory pool with 3 buffers of 16 bytes and 2 buffers of 64 bytes. If the size argument is 10, a buffer from the 16 byte pool will be allocated. After allocating two additional buffers of size 10, the 16 bytes buffer pool is exhausted. If allocating the fourth 10 bytes buffer, it will be allocated from the pool with the 64 bytes buffers. The next time that the function is called when all buffers of all sizes in the pool have been allocated, the function returns the value NULL to signify that no memory is available. The function also returns NULL if no buffers of the requested buffer size exists in the pools. For example, if the size argument is 70, then none of the pools in the example will match the request and the return value is NULL.

Example

```
myTypeA_t *pMsg = MM_AllocPool(myPools, sizeof(myTypeA_t));
```

2.4.6 MSG_Free

Macro Definition

```
#define MSG_Free(buffer) MM_Free(buffer)
```

Buffer

Specifies the buffer to be returned to the memory pool.

Functional Description

The macro is typically translated directly to MM_Free(size). See Section 2.4.7, “MM_Free” for more information.

2.4.7 MM_Free

Prototype

```
void MM_Free(void *pBuffer);
```

pBuffer

The buffer to be returned to the memory pool from which it was allocated.

Functional Description

The pBuffer argument points at a memory buffer allocated by the MSG_Alloc and MSG_AllocType macros, or the MM_Alloc and MM_AllocPool functions. It is important that the value of the pBuffer pointer is the same as the value returned by any of the four allocation methods. Otherwise, the memory pool will be invalidated.

Example

```
uint8_t *pMsg = MM_Alloc(14);
/* Important: Do not modify the value returned by any allocation function. E.g. pMsg++, or pMsg
+= startOfDataIndex etc. will cause the MM_Free function to invalidate the memory pool. */
MM_Free(pMsg);
```

2.4.8 MSG_Send

Macro Definition

```
#define MSG_Send(sap, msg)  (sap##_SapHandler((void *)(msg)))
```

Sap

Specifies the name of the service access point to which the message is sent.

Msg

Specifies the message to be send to the service access point.

Functional Description

The macro translates into a function with the name of the specified SAP concatenated with the extension "_SapHandler". For example, a network layer or application must implement a function called MLME_NWK_SapHandler. The 802.15.4 MAC will call that function when sending messages to the network layer or application.

The following SAPs must be defined by the upper layer:

```
MLME_NWK_SapHandler()
MCPS_NWK_SapHandler()
ASP_APP_SapHandler()
```

See Section 2.1, "General MAC/Network Interface Information", and Section 5.1, "General APP/ASP Interface Information" for more information about the service access points.

Example

```
mlmeMessage_t *pMsg = MSG_AllocType(mlmeMessage_t);
PrepareAssociateMessage(pMsg); // Pseudo code.
// Call NWK_MLME_SapHandler(pMsg);
MSG_Send(NWK_MLME, pMsg);
```

2.4.9 MSG_InitQueue

Macro Definition

```
#define MSG_InitQueue(anchor) List_ClearAnchor(anchor)
```

Anchor

Specifies a pointer to the queue anchor that must be initialized.

Functional Description

This macro translates to a function which clears the head and tail pointers of the queue anchor. All anchors must be cleared during system initialization before they are used for queuing messages.

Example

```
anchor_t myNwkQueue;
void MyApp_Init(void)
{
  MSG_InitQueue(&myNwkQueue);
}
```

2.4.10 MSG_Queue

Macro Definition

```
#define MSG_Queue(anchor, msg) List_AddTail((anchor), (msg))
```

Anchor

Specifies a pointer to the queue anchor which will receive the message.

Msg

Specifies a pointer to the message being queued.

Functional Description

The macro translates to a function which adds the message to the tail of the queue. There is no limit imposed on the number of messages that can be stored in the queue. When combined with the MSG_DeQueue macro, a FIFO queue is realized.

Example

```
anchor_t myMlmeNwkQueue;

void MLME_NWK_SapHandler(nwkMessage_t *pMsg)
{  // Queue up messages from the MLME.
  MSG_Queue(&myMlmeNwkQueue, pMsg);
}
```

2.4.11 MSG_QueueHead

Macro Definition

```
#define MSG_QueueHead(anchor, msg) List_AddHead((anchor), (msg))
```

Anchor

Specifies a pointer to the queue anchor which will receive the message.

Msg

Specifies a pointer to the message being queued.

Functional Description

This macro translates to a function which adds the message to the head of the queue. There is no limit imposed on the number of messages that can be stored in the queue. When combined with the MSG_DeQueue macro a LIFO queue, or stack, is realized because messages are always de-queued from the head of the queue. See Section 2.4.12, “MSG_DeQueue”.

Example

```
anchor_t myStackQueue;

void AddToStack(myDataType_t *pMsg)
{ // Put the message on the stack-like queue
  MSG_QueueHead(&myStackQueue, pMsg);
}
```

2.4.12 MSG_DeQueue

Macro Definition

```
#define MSG_DeQueue(anchor) List_RemoveHead(anchor)
```

Anchor

Specifies a pointer to the queue anchor.

Returns

Pointer to de-queued message or NULL if queue is empty.

Functional Description

The macro translates to a function which removes a message from the head of the specified queue.

Example

```
anchor_t myMcpsNwkQueue;

void ProcessMcpsMessage(void)
{
  mcpsToNwkMessage_t *pMsg = MSG_DeQueue(&myMcpsNwkQueue);
  // Do something with the message.
  ...
  // ALWAYS free message when done with it.
  MSG_Free(pMsg);
```

2.4.13 Message Tracking

The message tracking feature allows monitoring the status of each buffer in the memory pool. The user can know the number of times a buffer was allocated, deallocated, its allocation status and the address of the function that last allocated or deallocated it.

The message tracking is enabled by setting MsgTracking_d on 1 in file AppToMacPhyConfig.h. Each buffer in the memory pool is associated with an element of type MsgTracking_t, stored in the MsgTrackingArray, which is defined in GlobalVars.c. The MsgTracking_t type definition is shown below.

```
typedef struct MsgTracking_tag {
  uint16_t MsgAddr;
  uint16_t AllocAddr;
  uint16_t FreeAddr;
  uint8_t  AllocCounter;
```

```
  uint8_t  FreeCounter;
  uint8_t  AllocStatus;
} MsgTracking_t;
```

The meaning of each field is described below:

- The MsgAddr field specifies the address of the buffer;
- The AllocAddr and FreeAddr fields specify the addresses where the buffer was allocated and respectively deallocated the last time;
- The AllocCounter and FreeCounter fields show the number of times the buffer was allocated and deallocated, respectively;
- The AllocStatus field specifies if the buffer is currently allocated (1) or deallocated (0).

Another useful feature allows to detect whenever the message system tries to allocate or deallocate an invalid buffer, e.g. a buffer that is outside the pool memory. The NoOfWrongAddrs increments each time the buffer accessed by MM_Alloc or MM_Free is not in the memory pool.

2.4.14 MSG_Pending

Macro Definition

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

Anchor

Specifies a pointer to the queue anchor.

Returns

TRUE if one or more messages in queue, or FALSE if queue is empty.

Functional Description

This macro checks if there are any messages in the specified queue. This is accomplished without disabling interrupts, and it is the recommended method for checking queues for messages. The MSG_DeQueue macro translates to a function which disables interrupts for a few micro seconds. Thus, if checking the queues in a tight loop, it is preferred to use MSG_Pending so as not to interfere with the real-time functionality of the MAC.

Example

```
anchor_t myMcpsNwkQueue;
while(running) {
  if(MSG_Pending(&myMcpsNwkQueue)) {
    // Function uses MSG_DeQueue to obtain queued message.
    ProcessMcpsMessage();
  }
}
```

Chapter 3
Interfacing to the 802.15.4 MAC Software

This chapter describes how to interface an application to the MAC and how to use the MAC interface functions. The examples shown in this chapter explain the API functions and are often simplified for this purpose. Refer to *802.15.4 MyWirelessApp Software User's Guide* (802154MWASUG) for a detailed walk-through of how to create a more advanced application.

A brief overview of the MAC interfaces are given in Section 3.1, "Interface Overview" before each interface is explained in more detail. Throughout this chapter, the term "application" refers to the next higher layer and all layers above the MAC layer. This could be the ZigBee Network, an application, or another network layer written directly on top of the MAC.

3.1 Interface Overview

The interface between the application layer and the MAC layer is based on service primitives passed as messages from one layer to another. These service primitives are implemented as a number of C-structures with fields for command opcodes/identifiers and command parameters. This chapter does not describe the primitives in detail but focuses on the functions used for passing, receiving, and processing the message primitives. For a description of all available message primitives see *Chapter 2, "MAC/Network Layer Interface Description"*.

Figure 3-1 shows the three interfaces to the MAC.

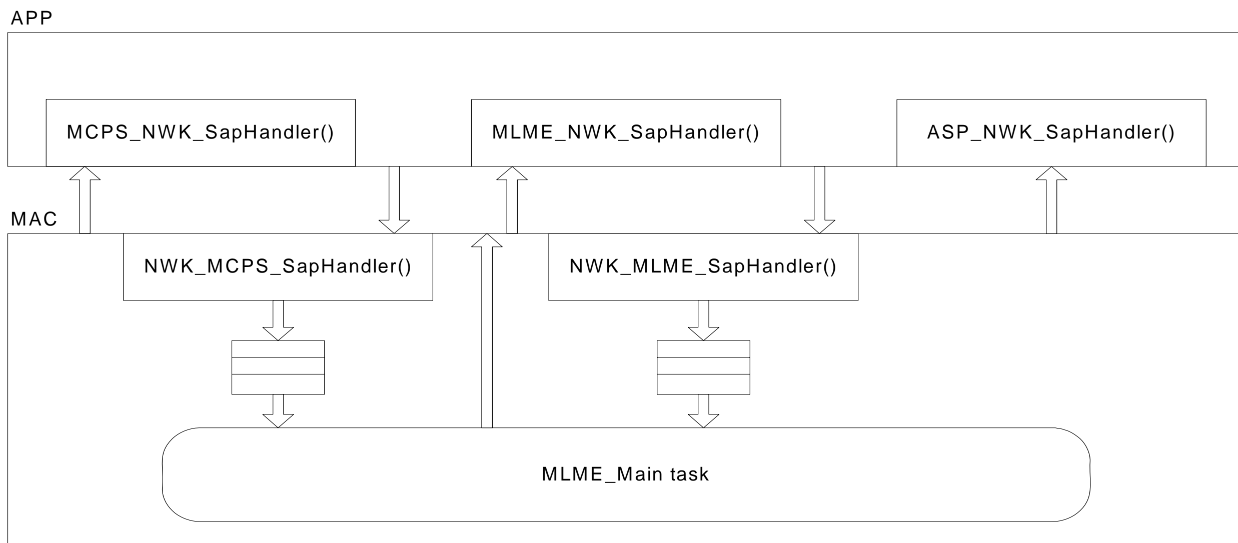

Figure 3-1. MAC Interfaces

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

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

1. `NWK_MLME_SapHandler()` – Used for command related primitives sent from the application (or network) layer to the MAC layer. This SAP receives all MLME request and response primitives. See Section 3.5, "MAC Main Task" for a detailed description.
2. `NWK_MCPS_SapHandler()` – Used for data related primitives sent from the application layer to the MAC layer. This SAP receives all MCPS request and response primitives. See Section 3.5, "MAC Main Task" for a detailed description.

The following list shows the three Service Access Points (SAPs) which must be implemented in the application layer by the MAC user.

1. `MLME_NWK_SapHandler()` – Used for command related primitives sent from the MAC layer to the application/network layer. The access point receives all MLME confirm and indication primitives. See Section 3.6, "MLME and MCPS Interface" for a detailed description.
2. `MCPS_NWK_SapHandler()` – Used for data related primitives sent from the application layer to the MAC layer. This SAP receives all MCPS confirm and indication primitives. See Section 3.6, "MLME and MCPS Interface" for a detailed description.
3. `ASP_APP_SapHandler()`– Used for receiving all ASP indications from the MAC. See Section 3.7, "ASP Interface" for a detailed description.

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

3.1.1 MC1310x, MC1320x, and MC1321x Transceiver IRQ Timing Dependency

For these devices, the transceiver is a separate device from the MCU. Parts of the MAC are implemented through a synchronous, interrupt driven structure (from the transceiver to the MCU). This area is referred to as the "transceiver ISR" because the controlling interrupts are generated by the MC1319x/MC1320x/MC1321x transceiver. These transceiver interrupts typically indicate events such as Rx data received, Rx timeouts, Tx done indications, and others. The most timing critical parts of the MAC are serviced through these interrupts. The following timing constraint must be kept by the application in order to meet MAC interrupt latency demands:

- Within a period of 64 µs, the application may only disable the transceiver interrupts for a maximum duration of 10 µs.

3.1.2 MC1322x Transceiver IRQ Timing Dependency

The MC1322x has the transceiver attached as a peripheral directly to the MCU bus, and also has a separate MAC controller and DMA for moving data. As such, the CPU typically need not respond to the transceiver on any periodic basis and does not have a similar timing constraint for handling interrupts.

3.2 Include Files

Table 3-1 shows which MAC files must be included in the application C-files in order to have access to the entire MAC API.

Table 3-1. Required MAC Include Files in Application C-Files

Include File Name	Description
EmbeddedTypes.h	Provides Freescale specific type defines for creating fixed sized variables. For example an uint8_t defines the type of an 8 bit unsigned integer. Also defines the TRUE and FALSE constants.
MsgSystem.h	Provides access to message handling functions and to allocation/deallocation of data and command buffers to/from the MAC/PHY.
NwkMacInterface.h	Defines structures and constants for all MLME and MCPS primitives.
AppAspInterface.h	Defines structures and constants for all ASP primitives.
PublicConst.h	Contains the 802.15.4 specific status/return codes.
PlatformInit.h	Platform initialization functions.
IrqControlLib.h	Provides primitives for interrupt protection, configuration, checking and acknowledgement.
FunctionLib.h	Generic function library.
TS_Interface.h	Provides the interface functions to the task scheduler interface.
TMR_Interface.h	The timer driver interface.
UartUtil.h	Needs to be included if the application uses the UART interface. Contains support functions for different UART operations. Functions like UartUtil_Print and UartUtil_PrintHex are defined here.

3.3 Source Files

Table 3-2 shows which MAC source files must be included in the application project.

Table 3-2. Required MAC Source Files in Application Project

Source File Name	Description
GlobalVars.c	Provides MAC specific variables and memory allocations.

3.4 MAC API

The MAC API provides a simple and consistent way of interfacing to the Freescale 802.15.4 MAC software. The number of API functions that the Freescale MAC software exposes to the application, are limited in order to keep the interfaces as simple and consistent as possible. The API functions available are used for initialization of the MAC, running the MAC, allocating, deallocating, and sending messages, and queueing and dequeueing of messages. Table 3-3 shows the available API functions for initializing and running the MAC.

Table 3-3. Available API Functions

Function	Description
Init_802_15_4()	This function initializes internal variables of the MAC/PHY modules, resets state machines etc. Once the function has been called the MAC layer services are available and the MAC and PHY layers are in a known and ready state for further access. The function is only available if the preprocessor definition INCLUDE_802_15_4 has been setup in the compiler settings of the MCP project.
Mlme_Main task	As the MAC has been designed to be independent of an operating system, part of the MAC must be executed regularly by the MAC task. This task has basically processes data/command messages that are pending in any of the MAC input queues. See Section 3.5, “MAC Main Task” for an in-depth description of the Mlme_Main() function.

For allocating and deallocating messages to and from the MAC and sending messages to the MAC, the MAC exposes the following message handling functions as shown in Table 3-4.

Table 3-4. Exposed Message Handling Functions

Function	Description
*pMsg = MSG_AllocType(type)	Allocate a message of a certain type. This must only be used to allocate messages for one of the MAC access points. The type parameter can be set to mlmeMessage_t, mcpsMessage_t, and aspMessage_t.
MSG_Free(*pMsg)	Frees a message that was allocated using MSG_AllocType().
status = MSG_Send(SAP, *pMsg)	Sending a message is equal to calling a Service Access Point function. The SAP argument can be either NWK_MLME or NWK_MCPS. The argument is translated into the corresponding SAP handler function, e.g. NWK_MLME_SapHandler().

The MAC implements a few functions for queueing and dequeuing messages from the MAC to the application. These functions are shown in Table 3-5.

Table 3-5. Queueing and Dequeuing Functions

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

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

Table 3-6. Data Structures Passed From The Application

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

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

Table 3-7. Data Structures Passed From The MAC

Data Type	Description
nwkMessage_t	The data structure of the messages passed from the MLME to the applications MLME SAP handler.
mcpsToNwkMessage_t	The data structure of the messages passed from the MCPS to the applications MCPS SAP handler.
aspToAppMsg_t	The data structure of the messages passed from the ASP to the applications ASP SAP handler.

Additionally, a helper structure for managing message queues is also defined.

Table 3-8. Helper Structures For Managing Message Queues

Data Type	Description
anchor_t	A container for any type of MAC message. Before queuing or dequeuing messages into the structure the anchor must be initialized using `MSG_InitQueue()`. Messages are queued and dequeued using `MSG_Queue()` and `MSG_DeQueue()`.

3.5 MAC Main Task

Because the MAC is designed to be independent of an operating system, part of the MAC is executed by the MAC task with the root function Mlme_Main. The MAC task is responsible for the following functions.

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

Even though these activities are not highly time critical in nature, the application program must execute this function in a timely manner. The specific requirements for this execution are as follows.

- The MAC main task must be executed at least once for every Rx packet the application wishes to receive. If the MAC is in a state where the receiver is enabled either continuously or with a regular interval, packets can be expected “any time” and the worst-case packet receive latency is increased with the interval, with which the application executes the Mlme_Main() function. The MAC will not crash if all receive buffers fill up due to slow Mlme_Main() polling, but instead the receiver will be switched off even though it should actually have been enabled
- The function must be executed at least once for every MCPS, MLME or ASP primitive the application has sent to one of the MAC access points
- In addition to the already stated requirements, the function must be executed at least once between two beacon receptions in order to ensure basic beacon operation. If this requirement is not met, beacon packets are not processed in a timely manner, which could lead to unexpected behavior. For optimal beacon operation it is recommended that the MAC main task is executed at least twice during every superframe

3.6 MLME and MCPS Interface

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

Common for the MCPS and MLME interface is that messages are sent to the interfaces using the MSG_Send(SAP, *pMsg) function (see Section 3.3, “Source Files”). If sending to the MLME the SAP parameter must be set to NWK_MLME and sending to the MCPS the SAP parameter must be set to NWK_MCPS.

3.6.1 Resetting

Before the Freescale 802.15.4 MAC layer can be accessed after power-on, it must be initialized by calling the Init_802_15_4() function. See Section 3.3, “Source Files” for a more detailed explanation.

At any point after this, it is always safe to reset the MAC (and also the PHY) layer by using the service MLME-RESET.request as shown in the following example code:

```
void App_ResetMac_Example(void)
{
  mlmeMessage_t mlmeReset;

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

By calling this service, the MAC layer is reset and brought into the same state, as it was right after having called Init_802_15_4(). The setDefaultPib parameter tells the MAC layer whether its PIB attributes should be set to their default value or if they are to stay unchanged after the reset. This is specified in the 802.15.4 Standard.

Notice that the reset is processed immediately (in the NWK_MLME_SapHandler()) and that the Freescale MAC does not generate the confirmation message, MLME-RESET.confirm. Furthermore there is no need to check for the return value of MSG_Send() as it always returns gSuccess_c on an MLME-RESET.request.

Since the call is processed immediately, the message structure need not be allocated through MSG_AllocType(), but can be allocated on the stack. In all circumstances, it is the responsibility of the calling entity to de-allocate the message for MLME-RESET.request.

3.6.2 Accessing PIB Attributes

The MAC PIB holds all the MAC attributes/variables that are accessible for the application. According to the 802.15.4 Standard, the primitives MLME-SET.request and MLME-GET.request, provide access to the PIB.

Similar to the Freescale implementation of MLME-RESET, these primitives are processed immediately and therefore the corresponding confirm messages MLME-SET.confirm and MLME-GET.confirm are not generated. Instead, the return code from MSG_Send() must be used to check the status. The MLME-SET.request message contains a pointer to the data to be written to the MAC PIB, which must be supplied by the application. The following code is an example of how to use MLME-SET.request.

```
uint8_t App_SetMacPib_Example(uint8_t attribute, uint8_t *pValue)
{
  mlmeMessage_t mlmeSet;

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

  return MSG_Send(NWK_MLME, &mlmeSet);
}
```

This usage is very similar to MLME-RESET.request, because the call is processed immediately. Therefore the message structure need not be allocated through MSG_AllocType(), but can be allocated. For example, it can allocated on the stack. It is the responsibility of the calling entity to de-allocate the message (and potentially the PIB attribute value data buffer) for MLME-SET.request.

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

3.6.3 MLME Primitives

The MLME-SET.request, MLME-GET.request, and MLME-RESET.requests are the only messages that are completed synchronously all other messages from the application to the MLME interface are completed asynchronously i.e. a confirmation message will be generated in the MLME and sent to the MLME SAP handler of the application (MLME_NWK_SapHandler()).

For example, in order to request an energy detection scan, the application must allocate a MLME message using MSG_AllocType(mlmeMessage_t) and fill out the parameters of the message (MLME-SCAN.request) and send the message to the MLME SAP handler as shown in the following code example. For a detailed explanation of energy detection scan, see Section 4.2.2, “Energy Detection Scan”.

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

  /* Allocate a message for the MLME. */
  pMsg = MSG_AllocType(mlmeMessage_t);
  if(pMsg != NULL)
  {
    /* Allocation succeeded. Fill out the message */
    pMsg->msgType = gMlmeScanReq_c;
    pScanReq->msgData.scanReq.scanType = gScanModeED_c;
    pScanReq->msgData.scanReq.scanChannels[0] = 0x00;
    pScanReq->msgData.scanReq.scanChannels[1] = 0xF8;
    pScanReq->msgData.scanReq.scanChannels[2] = 0xFF;
    pScanReq->msgData.scanReq.scanChannels[3] = 0x07;
    pScanReq->msgData.scanReq.scanDuration = 5;

    /* Send the Scan request to the MLME. */
    if(MSG_Send(NWK_MLME, pMsg) == gSuccess_c)
      return errorNoError;
    else
      return errorInvalidParameter;
  }
  else
  {
    /* Allocation of a message buffer failed. */
    return errorAllocFailed;
  }
}
```

When requesting a service that is completed asynchronously, it is the responsibility of the MLME to deallocate the message. In order for the application to be able to receive the MLME-SCAN.confirm message from the MLME (and for the application to be able to link without any errors) the application must implement the MLME SAP handler. This SAP handler will typically only queue the message (which is of type nwkMessage_t) and return as soon as possible. An event needs to be passed to the application task in order to notify it that a new message from the MLME has arrived. To tell the MLME that the message was received successfully the SAP handler must return a status code of gSuccess_c (any other return codes indicates a failure) as shown in the following code example.

```
uint8_t MLME_NWK_SapHandler(nwkMessage_t * pMsg)
{
  /* Put the incoming MLME message in the applications input queue. */
  MSG_Queue(&mMlmeNwkInputQueue, pMsg);
```

```
  TS_SendEvent(gZappTaskID_c, evtMessageFromMLME);
  return gSuccess_c;
}
```

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

```
void AppTask(event_t events)
{
   nwkMessage_t *pMsg;
   uint8_t *pEdList;

   /* Try to get a message from the MLME. */
   if (events & evtMessageFromMLME)
   {
     pMsg = MSG_DeQueue(&mMlmeNwkInputQueue);
     switch (pMsg->msgType)
    {
    /* Check for a scan confirm message. */
    case gNwkScanCnf_c:
      /* Find the pointer to the list of detected energies */
      pEdList = pMsg->msgData.scanCnf.resList.pEnergyDetectList;
      /* Do app. specific operation on the detected energies */
      … No code is shown for that
      /* The list of detected energies MUST be freed. */
      /* Note: This is a special exception for scan. */
      MSG_Free(pEdList);
      break;
    default:
      break;
    }
    /* Ensure to always free the message */
    MSG_Free(pMsg);
   }
}
```

Notice that the application must free the MLME message after having processed it and that some messages contain pointers to data structures that must be freed before freeing the message itself (as in the case shown in the previous example code). Neglecting to free such data structures (and the messages themselves) will cause memory leaks.

The MLME interface only allows for one request pending at a time. That is, after having sent a scan request message to the MLME, users must wait for a scan confirmation message on the MLME SAP handler until they are allowed to send another MLME request to the MLME. Not complying with this rule may result in unwanted and/or unpredictable behaviour.

If a MLME primitive is called with an invalid argument, the MAC will not pass it down to the lower layers, so the application will not have to wait for a confirmation. In this case MSG_Send will return the gInvalidParameter_c value. This approach simplifies the application state machine.

3.6.4 MCPS Primitives

Alongside the MLME interface, the MCPS interface processes data related messages (see Figure 3-1 in Section 3.1, “Interface Overview”). The MCSP interface is used just like the MLME interface. The main difference is the message types are sent back and forth on the interface. On the MCPS, users must use Msg_AllocType(gMaxRxTxDataLength_c) to allocate a MCPS message as shown in the following code example.

```
void SendMyName(void) {
    uint8_t FSL[] = "Freescale";
    nwkToMcpsMessage_t *pMsg;
   // DO LIKE THIS - remember to allocate room for data, structure only contains room for data
pointer and not the data.
    pMsg = MSG_Alloc(gMaxRxTxDataLength_c);
    if (pMsg) {
        pMsg->msgType = gMcpsDataReq_c;
        /* initialize pointer to Msdu */
        pMsg->msgData.dataReq.pMsdu = (uint8_t*)(&(pMsg->msgData.dataReq.pMsdu)) +
sizeof(uint8_t*);
        FLib_MemCpy(pMsg->msgData.dataReq.pMsdu, FSL, sizeof(FSL));
        FLib_MemCpy(pMsg->msgData.dataReq.dstAddr, maDeviceShortAddress, 2);
        FLib_MemCpy(pMsg->msgData.dataReq.srcAddr, (void *)maShortAddress, 2);
        FLib_MemCpy(pMsg->msgData.dataReq.dstPanId, (void *)maPanId, 2);
        FLib_MemCpy(pMsg->msgData.dataReq.srcPanId, (void *)maPanId, 2);
        pMsg->msgData.dataReq.dstAddrMode = gAddrModeShort_c;
        pMsg->msgData.dataReq.srcAddrMode = gAddrModeShort_c;
        pMsg->msgData.dataReq.msduLength = sizeof(Mads);
        /*Add Indirect option in order to send to polling end device*/
        pMsg->msgData.dataReq.txOptions = gTxOptsAck_c | gTxOptsIndirect_c;
        pMsg->msgData.dataReq.msduHandle = mMsduHandle++;
        /* Send the Data Request to the MCPS */
        (void)MSG_Send(NWK_MCPS, pMsg);
        /* Prepare for another data buffer */
    }
}
```

The application is allowed to allocate multiple data messages using MSG_AllocType(gMaxRxTxDataLength, ...) until this returns NULL. Unless the MAC is running in non-beacon mode as a device, it reserves a buffer for general receive and for transmitting beacons.

Because the MCPS interface can manage multiple outstanding data requests, it is possible to use double (or multiple) buffering for maximum throughput. That is, it is possible to send a data request to the MCPS and then, before receiving a data confirmation on that request, send another data request which keeps a constant data flow between the application and the MCPS interface. Even though it is not supported by the 802.15.4 Standard, double buffering can also be used for polling. See Section 4.7.1, “Data Primitives” for a description of how to optimize data throughput using double buffering.

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

```
uint8_t MCPS_NWK_SapHandler(mcpsToNwkMessage_t *pMsg)
{
  /* Put the incoming MCPS message in the applications input queue. */
  MSG_Queue(&mMcpsNwkInputQueue, pMsg);
```

```
  TS_SendEvent(gZappTaskID_c, evtMessageFromMCPS);
  return gSuccess_c;
}
```

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

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

3.7 ASP Interface

The ASP interface is a Freescale proprietary interface that can perform several functions including power management, access non-volatile (NV) RAM and others. Unlike the MCPS and MLME primitives, the ASP primitives are implemented as direct functions calls like in the code shown below.

```
void App_EnterHibernation_Example(void)
{
  uint8_t res = Asp_HibernateReq();
}
```

Notice that the application is responsible for de-allocating the messages that it sends to the ASP. However, as a consequence of the ASP receiving a command (for example, an Asp_DozeReq to send the MC 13192 into doze mode), the ASP may at some point send an indication message to the application (for example, an ASP-WAKE.indication when the MC1319x/MC1321x wakes up from doze mode). As was the case for the MLME and MCPS interfaces, the ASP sends back indication messages (of type aspToAppMsg_t) to the application using the applications ASP SAP handler ASP_APP_SapHandler()as shown in the following example code.

```
uint8_t ASP_APP_SapHandler(aspToAppMsg_t *pMsg)
{
  /* Put the incoming ASP message in the applications input queue. */
  MSG_Queue(&mAspAppInputQueue, pMsg);
  TS_SendEvent(gZappTaskID_c, evtMessageFromASP);
  return gSuccess_c;
}
```

NOTE

By queuing the ASP msg to the AppTask, the AppTask is responsible for calling MSG_Free(pMsg) to free the msg.

Again, the ASP messages are typically processed in the application task, just like the MLME and MCPS messages. It is the responsibility of the application to de-allocate the ASP message.

Chapter 4
Feature Descriptions

The chapter contains descriptions of the Freescale 802.15.4 MAC/PHY software features, focusing on the implementation specific details of both the 802.15.4 2003 and 2006 Standards. Refer to the appropriate 802.15.4 Standard for more details on these features.

NOTE

The differences relevant to the MAC software between the 802.15.4 Standard (2003) and the 802.15.4 Standard (2006) are noted where appropriate.

4.1 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.

4.1.1 PIB Attributes

Table 4-1 shows all the MAC PIB attributes available including Freescale specific additions to the 802.15.4 Standard specific attributes.

Table 4-1. Available PIB Attributes

PIB Attribute	Description
Freescale Specific Attributes	
0x20	gMPibRole_c — Contains the current role of the device: 0x00 = Device 0x01 = Coordinator 0x02 = PAN Coordinator
0x21	gMPibLogicalChannel_c — Contains the current logical channel (11 to 26).
0x22	gMPibTreemodeStartTime_c — Used to support Beaconed Tree Mode.
0x23	gMPibPanIdConflictDetection_c - Disables or enables PAN ID conflict detection.
0x24	gMPibBeaconResponseDenied_c - If set on TRUE, the coordinator will not respond to beacon requests. **Note:** This PIB attribute is available for ARM7 based MC1322x platform.

Table 4-1. Available PIB Attributes (continued)

PIB Attribute	Description
0x25	gMPibNBSuperFrameInterval_c - The length of the superframe for non-beacon mode. This attribute is used instead of gMPibBeaconOrder_c for the calculation of the persistence time of indirect packets, when the coordinator runs in non-beacon mode. See the IEEE™ 802.15.4 Standard for details on how the persistence time is calculated. **Note:** This PIB attribute is available for ARM7 based MC1322x platform.
0x27	gMacBeaconResponseLQIThreshold_c - Used to filter incoming beacon requests based on their LQI value. 0x00 (default) = filter disabled. The coordinator will respond to all incoming beacon requests. 0x01 - 0xFF = filter enabled. The coordinator will not respond to incoming beacon requests having the LQI smaller than the configured threshold. **Note:** This PIB attribute is not available for the ARM7 based MC1322x platform.
802.15.4 Specific Attributes (see [1] for descriptions)	
0x40	gMPibAckWaitDuration_c
0x41	gMPibAssociationPermit_c
0x42	gMPibAutoRequest_c
0x43	gMPibBattLifeExt_c
0x44	gMPibBattLifeExtPeriods_c
0x45	gMPibBeaconPayload_c
0x46	gMPibBeaconPayloadLength_c
0x47	gMPibBeaconOrder_c
0x48	gMPibBeaconTxTime_c
0x49	gMPibBsn_c
0x4A	gMPibCoordExtendedAddress_c
0x4B	gMPibCoordShortAddress_c
0x4C	gMPibDsn_c
0x4D	gMPibGtsPermit_c
0x4E	gMPibMaxCsmaBackoffs_c
0x4F	gMPibMinBe_c
0x50	gMPibPanId_c
0x51	gMPibPromiscuousMode_c
0x52	gMPibRxOnWhenIdle_c
0x53	gMPibShortAddress_c
0x54	gMPibSuperFrameOrder_c
0x55	gMPibTransactionPersistenceTime_c
0x56	gMPibAssociatedPANCoord_c (available only for MAC 2006)
0x57	gMPibMaxBe_c (available only for MAC 2006)

Table 4-1. Available PIB Attributes (continued)

PIB Attribute	Description
0x59	gMPibMacFrameRetries_c (available only for MAC 2006)
0x5A	gMPibResponseWaitTime_c (available only for MAC 2006)
0x5B	gMPibSyncSymbolOffset_c (available only for MAC 2006)
0x5C	gMPibTimeStampSupported_c (available only for MAC 2006)
0x5D	gMPibSecurityEnable_c (available only for MAC 2006)
Security Specific Attributes	
0x70	gMPibAclEntryDescriptorSet_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	gMPibAclEntryDescriptorSetSize_c — The number of entries in the ACL descriptor set. Size is 1 byte.
0x72	gMPibDefaultSecurity_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	gMPibDefaultSecurityMaterialLength_c — The number of bytes in the ACL Security Material. Size is 1 byte.
0x74	gMPibDefaultSecurityMaterial_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	gMPibDefaultSecuritySuite_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	gMPibSecurityMode_c — The identifier of the security mode in use. 0x00 = Unsecured mode, 0x01 = ACL mode, 0x02 is Secured Mode. Size is 1 byte.
0x71	gMPibKeyTable_c (available only for MAC 2006) - A table of KeyDescriptor entries, each containing keys and related information required for secured communications.
0x72	gMPibKeyTableEntries_c (available only for MAC 2006) - The number of entries in *macKeyTable.* This PIB is read-only.
0x73	gMPibDeviceTable_c (available only for MAC 2006) - A table of DeviceDescriptor entries, each indicating a remote device with which this device securely communicates.
0x74	gMPibDeviceTableEntries_c (available only for MAC 2006) - The number of entries in *macDeviceTable.* This PIB is read-only.
0x75	gMPibSecurityLevelTable_c (available only for MAC 2006) - A table of SecurityLevelDescriptor entries, each with information as to the minimum security level expected depending on incoming frame type and subtype.
0x76	gMPibSecurityLevelTableEntries_c (available only for MAC 2006) - The number of entries in *macSecurityLevelTable.*
0x77	gMPibFrameCounter_c (available only for MAC 2006) - The outgoing frame counter for this device.
0x78	gMPibAutoRequestSecurityLevel_c (available only for MAC 2006) - The security level used for automatic data requests.
0x79	gMPibAutoRequestKeyIdMode_c (available only for MAC 2006) - The key identifier mode used for automatic data requests. This attribute is invalid if the *macAutoRequestSecurityLevel* attribute is set to 0x00.

Table 4-1. Available PIB Attributes (continued)

PIB Attribute	Description
0x7A	`gMPibAutoRequestKeySource_c` (available only for MAC 2006) - The originator of the key used for automatic data requests. This attribute is invalid if the *macAutoRequestKeyIdMode* element is invalid or set to 0x00.
0x7B	`gMPibAutoRequestKeyIndex_c` (available only for MAC 2006) - The index of the key used for automatic data requests. This attribute is invalid if the *macAutoRequestKeyIdMode* attribute is invalid or set to 0x00.
0x7C	`gMPibAutoDefaultKeySource_c` (available only for MAC 2006) – The originator of the default key used for key identifier mode 0x01.
0x7D	`gMPibPANCoordExtendedAddress_c` (available only for MAC 2006) – The 64-bit address of the PAN coordinator.
0x7E	`gMPibPANCoordShortAddress_c` (available only for MAC 2006) – The 16-bit short address assigned to the PAN coordinator. A value of 0xfffe indicates that the PAN coordinator is only using its 64-bit extended address. A value of 0xffff indicates that this value is unknown.
0x7F	`gMPibKeyIdLookupDescriptor_c` (available only for MAC 2006) – A list of KeyIdLookupDescriptor entries used to identify this KeyDescriptor.
0x80	`gMPibKeyIdLookupEntries_c` (available only for MAC 2006) – The number of entries in KeyIdLookupList.
0x81	`gMPibKeyDeviceList_c` (available only for MAC 2006) – A list of KeyDeviceDescriptor entries indicating which devices are currently using this key, including their blacklist status.
0x82	`gMPibKeyDeviceListEntries_c` (available only for MAC 2006) – The number of entries in KeyDeviceList.
0x83	`gMPibKeyUsageList_c` (available only for MAC 2006) – A list of KeyUsageDescriptor entries indicating which frame types this key may be used with.
0x84	`gMPibKeyUsageListEntries_c` (available only for MAC 2006) – The number of entries in KeyUsageList.
0x85	`gMPibKey_c` (available only for MAC 2006) – The actual value of the key.
0x86	`gMPibKeyUsageFrameType_c` (available only for MAC 2006) – The type of the frame to be secured.
0x87	`gMPibKeyUsageCmdFrameId_c` (available only for MAC 2006) – The ID of the command frame to be secured.
0x88	`gMPibKeyDeviceDescriptorHandle_c` (available only for MAC 2006) – Handle to the DeviceDescriptor corresponding to the device.
0x89	`gMPibUniqueDevice_c` (available only for MAC 2006) – Indicator as to whether the device indicated by DeviceDescriptorHandle is uniquely associated with the KeyDescriptor, i.e. it is a link key as opposed to a group key.
0x8A	`gMPibBlackListed_c` (available only for MAC 2006) – Indicator as to whether the device indicated by DeviceDescriptorHandle previously communicated with this key prior to the exhaustion of the frame counter. If TRUE, this indicates that the device shall not use this key further, since it exhausted its use of the frame counter used with this key.
0x8B	`gMPibSecLevFrameType_c` (available only for MAC 2006) – The type of the frame to be secured.
0x8C	`gMPibSecLevCommandFrameIdentifier_c` (available only for MAC 2006) – The ID of the command frame to be secured.
0x8D	`gMPibSecLevSecurityMinimum_c` (available only for MAC 2006) – The minimal required/expected security level for incoming MAC frames with the indicated frame type and, if present, command frame type.

Table 4-1. Available PIB Attributes (continued)

PIB Attribute	Description
0x8E	gMPibSecLevDeviceOverrideSecurityMinimum_c (available only for MAC 2006) – Indicator as to whether originating devices for which the Exempt flag is set may override the minimum security level indicated by the SecurityMinimum element.
0x8F	gMPibDeviceDescriptorPanId_c (available only for MAC 2006) – The 16-bit PAN identifier of the device in this DeviceDescriptor.
0x90	gMPibDeviceDescriptorShortAddress_c (available only for MAC 2006) – The 16-bit short address of the device in this DeviceDescriptor. A value of 0xfffe indicates that this device is only using its extended address. A value of 0xffff indicates that this value is unknown.
0x91	gMPibDeviceDescriptorExtAddress_c (available only for MAC 2006) – The 64-bit IEEE extended address of the device in this DeviceDescriptor. This element is also used in unsecuring operations on incoming frames.
0x92	gMPibDeviceDescriptorFrameCounter_c (available only for MAC 2006) – The incoming frame counter of the device in this DeviceDescriptor. This value is used to ensure sequential freshness of frames.
0x93	gMPibDeviceDescriptorExempt_c (available only for MAC 2006) – Indicator as to whether the device may override the minimum security level settings.
0x94	gMPibKeyIdLookupData_c (available only for MAC 2006) – Data used to identify the key.
0x95	gMPibKeyIdLookupDataSize_c (available only for MAC 2006) – A value of 0x00 indicates a set of 5 octets; a value of 0x01 indicates a set of 9 octets.
Freescale Specific Security Attributes	
0x77	gMPibAclEntryCurrent_c — Sets which ACL entry is active for access (0 indicates first entry, 1 second entry and so on). Size is 1 byte.
0x78	gMPibAclEntryExtAddress_c — 64 bit addr of the device in this ACL entry. Size is 8 bytes.
0x79	gMPibAclEntryShortAddress_c — 16 bit addr of the device in this ACL entry. Size is 2 bytes.
0x7A	gMPibAclEntryPanId_c — PAN ID of the device in this ACL entry. Size is 2 bytes.
0x7B	gMPibAclEntrySecurityMaterialLength_c — Number of bytes in 'aclSecurityMaterial' (<=16). Size is 1 byte.
0x7C	gMPibAclEntrySecurityMaterial_c — Key for protecting frames. Size is 16 bytes.
0x7D	gMPibAclEntrySecuritySuite_c — Security suite used for the device in this ACL entry. Size is 1 byte.
0x96	gMPibKeyTableCrtEntry_c (available only for MAC 2006) – The current entry in the macKeyTable.
0x97	gMPibDeviceTableCrtEntry_c (available only for MAC 2006) – The current entry in the macDeviceTable.
0x98	gMPibSecurityLevelTableCrtEntry_c (available only for MAC 2006) – The current entry in the macSecurityLevelTable.
0x99	gMPibKeyIdLookupListCrtEntry_c (available only for MAC 2006) – The current entry in the KeyIdLooupList.
0x9A	gMPibKeyUsageListCrtEntry_c (available only for MAC 2006) – The current entry in the KeyUsageList.
0x9B	gMPibKeyDeviceListEntry_c (available only for MAC 2006) – The current entry in the KeyDeviceList.

4.1.2 Configuration Primitives

This section describes the implementation of the configuration related primitives.

4.1.2.1 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 and a confirm message is not 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;
```

4.1.2.2 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;
```

4.1.2.3 Set Request

The MLME-SET.request is used for modifying parameters in the MAC PIB. See Section 4.1, “Configuration” 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. A confirm message is not 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.

For MAC 2003 version the SetRequest structure is the following:

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

For MAC 2006 version the SetRequest structure is the following:

```
typedef struct mlmeSetReq_tag {
  uint8_t pibAttribute;
  uint8_t pibAttributeIndex;
  uint8_t *pibAttributeValue; // Pointer supplied by NWK
} mlmeSetReq_t;
```

`pibAttributeIndex` is not used.

4.1.2.4 Set Confirm

The Set-Confirm is not used because the Set-Request is carried out synchronously. See Section 4.1.2.3, "Set Request" for more information.

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

4.1.2.5 Get Request

The MLME-GET.request reads parameters in the MAC PIB. See Table 4-1 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. A confirm message is not generated. Instead, the return code from the MSG_Send() macro is used as the status code.

For MAC 2003 version the GetRequest structure is the following:

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

For MAC2006 version the GetRequest structure is the following:

```
typedef struct mlmeGetReq_tag {
  uint8_t  pibAttribute;
  uint8_t  pibAttributeIndex;
  uint8_t  *pibAttributeValue;  // Not in spec.
} mlmeGetReq_t;
```

`pibAttributeIndex` is not used.

4.1.2.6 Get Confirm

Get-Confirm is not used because the Get-Request is carried out synchronously. See Section 4.1.2.5, "Get Request" more information.

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

4.1.3 Configuration Examples

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

The following code snippet sends a Set-Request with a 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);
```

The following code snippet uses the stack instead of using the 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);
```

The following code snippet shows how to get the 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).
```

4.2 Scan Feature

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

NOTE

The Freescale MAC does not provide standalone PLME-ED primitives for energy detect measurement. Alternatively, the MLME_SCAN primitives are used where the scan can be limited to one channel.

4.2.1 Common Parts

Requesting any of the scan types (using the MLME-SCAN.request primitive) interrupts all other system activity at the MLME layer and below in accordance with the 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 are attempted and that at least one channel is always indicated in the ScanChannels parameter.

4.2.2 Energy Detection Scan

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

The MLME-SCAN.confirm primitive always holds energy detection results from all requested channels, that is, partial responses are never returned.

The level for Energy Detection is reported as required by the 802.15.4 Standard with an integer value from 0x00-0xFF. The hardware measured values are scaled and normalized for this range with the minimum value of 0x00 set to -100dBm and the maximum value of 0xFF set to -15 dBm. Measured values between -15dBm and -100dBm are scaled linearly between 0x00 and 0xFF.

The actual measured value is dependent on the target silicon:

- MC1319x, MC1320x and MC1321x families:
  - — The energy levels are reported by hardware and measured in ½ dBm steps. The hardware values range from 0x00 corresponding to about –80 dBm (theoretical minimal value) and 0xA0

(decimal 160) corresponding to 0 dBm (theoretical maximal value). Actual tests indicate a practical minimal value as 0x0A (-75dBm) and a maximal value as 0x82 (-15dBm).

— The MAC scales these hardware values to the proper reported ED value as stated above.

- MC1322x family:
  - — The energy level is mathematically derived from several reported hardware values. The values range from 0x00 (-100dBm) to 0xFF (-15dBm) as required.

4.2.3 Active and Passive Scan

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

Active and Passive Scan are capable of returning up to ten (10) results in a single MLME-SCAN.confirm primitive. Thus, when ten (10) unique beacons (see the 802.15.4 Standard) are received, the Scan is terminated in accordance with the 802.15.4 Standard, even if all channels have not been scanned to completion.

The pan descriptors are grouped by 5 (five) in 2 (two) linked blocks. The pPanDescriptorBlocks field of the scan confirmation message points to the first block.Each block contains the pointer to the first PAN descriptor in the block and the number of PAN descriptors in that block.The block structure is as follows:

```
struct panDescriptorBlock_tag {
  panDescriptor_t descriptorList[aScanResultsPerBlock];
  uint8_t descriptorCount;
  struct panDescriptorBlock_tag *pNext;
};
```

4.2.4 Orphan Scan

When Orphan Scan is requested, the device waits 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 ignores the command and re-enters Rx mode, as long as there is time for the shortest possible Rx cycle to complete before the complete scan time has elapsed.

If a valid coordinator realignment response is received while performing the Orphan Scan, scanning is immediately terminated in accordance with the 802.15.4 Standard [1], 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.

4.2.5 Scan Primitives

This section describes the implementation of the Scan related primitives.

4.2.5.1 Scan Request

The Scan-Request message parameters are directly mapped to the message parameters listed and described in the 802.15.4 Standard [1]. Users must ensure that scanChannels always indicates at least one valid channel and that channels outside the valid range [11 through 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].

For MAC 2003 version the ScanRequest structure is the following:

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

For MAC 2006 version the ScanRequest structure is the following:

```
typedef struct mlmeScanReq_tag {
  uint8_t  scanType;
  uint8_t  scanChannels[4];
  uint8_t  scanDuration;

  uint8_t  securityLevel;
  uint8_t  keyIdMode;
  uint8_t  keySource[8];
  uint8_t  keyIndex;
} mlmeScanReq_t;
```

- securityLevel — The security level to be used.
- keyIdMode — This mode identifies the key to be used. This parameter is ignored if the securityLevel parameter is set to 0.
- keySource — The originator of the key to be used. This parameter is ignored if the keyIdMode parameter is ignored or set to 0x00.
- keyIndex — The index of the key to be used. This parameter is ignored if the keyIdMode parmeter is ignored.

4.2.5.2 Scan-Confirm

The Scan-Confirm structure contains a pointer to an array of blocks containing PAN descriptors or a pointer to an array of energy levels. See the definition in Section 4.2.5.6, “PAN Descriptor”. 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 {
    uint8_t *pEnergyDetectList;                    // pointer to 16 byte static buffer
   panDescriptorBlock_t *pPanDescriptorBlocks; // this one must be freed by the upper layer
```

```
  } resList;
} nwkScanCnf_t;
```

4.2.5.3 Orphan Indication

For MAC 2003 version the OrphanIndication structure is the following:

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

For MAC 2006 version the OrphanIndication structure is the following:

```
typedef struct nwkOrphanInd_tag {
  uint8_t  orphanAddress[8];
  uint8_t  securityLevel;
  uint8_t  keyIdMode;
  uint8_t  keySource[8];
  uint8_t  keyIndex;
} nwkOrphanInd_t;
```

The security parameter has the same meaning as the ScanRequest security parameters.

4.2.5.4 Orphan Response

For MAC 2003 version the OrphanResponse structure is the following:

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

For MAC 2006 version the OrphanResponse structure is the following:

```
typedef struct mlmeOrphanRes_tag {
  uint8_t  orphanAddress[8];
  uint8_t  shortAddress[2];
  bool_t   associatedMember;

  uint8_t  securityLevel;
  uint8_t  keyIdMode;
  uint8_t  keySource[8];
  uint8_t  keyIndex;
} mlmeOrphanRes_t;
```

The security parameter has the same meaning as the ScanRequest security parameters.

4.2.5.5 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 because it contains pointers. The pAddrList pointer points to the address list which is formatted according to the 802.15.4 Standard. The pPanDescriptor pointer points to the pan descriptor of the indication message. See the definition in Section 4.2.5.6, “PAN Descriptor”. The pSdu is the beacon payload buffer. The pBufferRoot pointer contains the data fields pointed to by the other pointers, and is used for freeing only.

WARNING

The pBufferRoot must be freed before the indication message is freed. 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;
```

4.2.5.6 PAN Descriptor

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

NOTE

Link Quality Indication (LQI) is used as part of the PAN descriptor structure representing an integer value from 0x00-0xFF where 0x00 equates to -100dBm and the maximum value of 0xFF to -15 dBm.

For MAC 2003 version the PanDescriptor structure is the following:

```
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;
```

For MAC 2006 version the PanDescriptor structure is the following:

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

The security parameter has the same meaning as the ScanRequest security parameters.

4.3 Start Feature

The start feature is not supported on RFD type devices and derivatives. According to the 802.15.4 Standard [1], 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) is returned.

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

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

4.3.1 Start Primitives

This section describes the implementation of the Start related primitives.

4.3.1.1 Start Request

Before sending a Start-Request, the macShortAddress must be set to something other than 0xFFFF.

For MAC 2003 the StartRequest structure is the following:

```
// 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;
```

For MAC 2006 the StartRequest structure is the following:

```
typedef struct mlmeStartReq_tag {
  uint8_t  panId[2];
  uint8_t  logicalChannel;
  uint32_t  startTime;
  uint8_t  beaconOrder;
  uint8_t  superFrameOrder;
  bool_t   panCoordinator;
  bool_t   batteryLifeExt;
  bool_t   coordRealignment;

  uint8_t  coordRealignSecurityLevel;
  uint8_t  coordRealignKeyIdMode;
  uint8_t  coordRealignKeySource[8];
  uint8_t  coordRealignKeyIndex;

  uint8_t  beaconSecurityLevel;
  uint8_t  beaconKeyIdMode;
  uint8_t  beaconKeySource[8];
  uint8_t  beaconKeyIndex;
} mlmeStartReq_t;
```

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

4.3.1.2 Start Confirm

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

4.4 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 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 is received, the macAutoRequest PIB attribute can be set to TRUE again. If consecutive beacons (aMaxLostBeacons) are lost, the MAC sends an MLME-SYNC-LOSS.indication.

NOTE

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

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

4.4.1 Synchronization Primitives

This section describes the implementation of the synchronization related primitives.

4.4.1.1 Sync Request

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

4.4.1.2 Sync Loss Indication

For MAC 2003 version the SyncLossIndication structure is the following:

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

For MAC 2006 version the SyncLossIndication structure is the following:

```
typedef struct nwkSyncLossInd_tag {
  uint8_t  lossReason;
  uint8_t  panId[2];
  uint8_t  logicalChannel;

  uint8_t  securityLevel;
  uint8_t  keyIdMode;
  uint8_t  keySource[8];
  uint8_t  keyIndex;
} nwkSyncLossInd_t
```

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

4.5 Association Feature

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

1. The association request itself.
2. The data request that is sent after aResponseWaitTime symbols.

Refer to the 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 802.15.4 Standard
- phyLogicalChannel — This attribute is updated as required by the 802.15.4 Standard
- macCoordExtendedAddress — This attribute is updated if the extended address of the coordinator is passed as argument. Otherwise it is not affected

- macCoordShortAddress — This attribute is updated with the value passed as argument if short addressing mode is used. This is stated in the 802.15.4 Standard. If the extended coordinator address is used in the call it is not possible to update this attribute – the short address of the coordinator is unknown. The 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 802.15.4 Standard, it should not violate the intention of the 802.15.4 Standard

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

The timeout value has a different meaning 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 already stated is used if the beacon is not being tracked. If the beacon is being tracked, it implies that the timeout value corresponds to CAP symbols. In this case, the timeout can extend over several superframes

WARNING

If the beacon is being tracked, there are some implications that may not be readily apparent. For example, consider a superframe configuration with macBeaconOrder = 14 and macSuperframeOrder = 0. In this example, the CAP will be approximately 900 symbols (depending on the length of the beacon frame). Beacons will be transmitted approximately every 4 minutes. The aResponseWaitTime is equal to 30.720 symbols. This implies that the timeout will occur after approximately 34 superframes, which in this example, is more than two hours.

Always pay close attention to the impact of the aResponseWaitTime value as it relates to association requests.

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. 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 802.15.4 Standard [1]. 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 frames (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 follows:

- 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 802.15.4 Standard [1]. The short address of the coordinator is not present in the response

An MLME-COMM-STATUS.indication message is generated on the coordinator when the response has been extracted by the device.

4.5.1 Association Primitives

This section describes the implementation of the Association related primitives.

4.5.1.1 Associate Request

Before sending the Associate-Request primitive, the 802.15.4 Standard [1] states that a Reset-Request must be sent, and an Active or Passive Scan was performed.

For MAC 2003 version the AssociateRequest structure is the following:

```
// 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;
```

For MAC 2006 version the AssociateRequest structure is the following:

```
typedef struct mlmeAssociateReq_tag {
  uint8_t  coordAddress[8];
  uint8_t  coordPanId[2];
  uint8_t  coordAddrMode;
  uint8_t  logicalChannel;

  uint8_t  securityLevel;
  uint8_t  keyIdMode;
  uint8_t  keySource[8];
```

```
  uint8_t  keyIndex;

  uint8_t  capabilityInfo;
} mlmeAssociateReq_t;
```

The security parameter has the same meaning as the ScanRequest security parameters.

4.5.1.2 Associate Response

For MAC 2003 version the AssociationResponse structure is the following:

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

For MAC 2006 version the AssociationResponse structure is the following:

```
typedef struct mlmeAssociateRes_tag {
  uint8_t  deviceAddress[8];
  uint8_t  assocShortAddress[2];
  uint8_t  securityLevel;
  uint8_t  keyIdMode;
  uint8_t  keySource[8];
  uint8_t  keyIndex;
  uint8_t  status;
} mlmeAssociateRes_t;
```

The security parameter has the same meaning as the ScanRequest security parameters.

4.5.1.3 Associate Indication

For MAC 2006 version the AssociateIndication structure is the following:

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

For MAC 2006 version the AssociateIndication structure is the following:

```
typedef struct nwkAssociateInd_tag {
  uint8_t  deviceAddress[8];
  uint8_t securityLevel;
  uint8_t keyIdMode;
  uint8_t keySource[8];
  uint8_t keyIndex;
  uint8_t  capabilityInfo;
} nwkAssociateInd_t;
```

- securityLevel — The security level purportedly used by the received MAC command frame.

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

4.5.1.4 Associate Confirm

For MAC 2003 version the AssociateConfirm structure is the following:

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

For MAC 2006 version the AssociateConfirm structure is the following:

```
typedef struct nwkAssociateInd_tag {
  uint8_t  deviceAddress[8];

  uint8_t securityLevel;
  uint8_t keyIdMode;
  uint8_t keySource[8];
  uint8_t keyIndex;

  uint8_t  capabilityInfo;
} nwkAssociateInd_t
```

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

4.5.2 Associate Example

This section shows an example of sending an associate-request. Some pseudo-code has been used (the AssociateFillInParms(),and HandleAssocConf() functions do 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.
}
```

4.6 Disassociation Feature

Disassociation is less complex than association, but there is an issue in the 802.15.4 Standard that makes disassociation from a coordinator difficult in a non-beacon enabled PAN network so special care must be taken when disassociating from a coordinator in a non-beacon 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.

The 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 so 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).

- 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.

4.6.1 Disassociation Primitives

This section describes the implementation of the Disassociation related primitives.

4.6.1.1 Disassociate Request

For MAC 2003 version the DisassociateRequest structure is the following:

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

For MAC 2006 version the DisassociateRequest structure is the following:

```
typedef struct mlmeDisassociateReq_tag {
  uint8_t  deviceAddress[8];
  uint8_t  devicePanId[2];
  uint8_t  deviceAddrMode;
  uint8_t  disassociateReason;
  bool_t   txIndirect;
  uint8_t  securityLevel;
  uint8_t  keyIdMode;
  uint8_t  keySource[8];
  uint8_t  keyIndex;
} mlmeDisassociateReq_t;
```

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

Security parameters have the same meaning as ScanRequest security parameters.

4.6.1.2 Disassociate Indication

For MAC 2003 version the DisassociateIndication structure is the following:

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

For MAC 2006 version the DisassociateIndication structure is the following:

```
typedef struct nwkDisassociateInd_tag {
  uint8_t  deviceAddress[8];
  uint8_t  disassociateReason;

  uint8_t  securityLevel;
  uint8_t  keyIdMode;
  uint8_t  keySource[8];
  uint8_t  keyIndex;
} nwkDisassociateInd_t
```

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

4.6.1.3 Disassociate Confirm

For MAC 2003 version the DisassociateConfirm structure is the following:

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

For MAC 2006 version the DisassociateConfirm structure is the following:

```
typedef struct nwkDisassociateCnf_tag {
  uint8_t deviceAddress[8];
  uint8_t devicePanId[2];
  uint8_t deviceAddrMode;
  uint8_t  status;
} nwkDisassociateCnf_t;
```

- deviceAddress — The address of the device that has either requested disassociation or been instructed to disassociate by its coordinator.
- devicePanId — The PAN identifier of the device that has either requested disassociation or been instructed to disassociate by its coordinator.

- deviceAddrMode — The addressing mode of the device that has either requested disassociation or been instructed to disassociate by its coordinator.

4.7 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 mainly used for holding Tx or Rx packets and they are limited to a specific number for each Device Type.

Table 4-2 shows the number of available large data buffers. The numbers vary because the different Device Types have different levels of functionality.

Table 4-2. Available Large Data Buffers

Device type	RFD	RFDNB	RFDNBNS	FFD	FFDNGTS	FFDNB	FFDNBNS
Bufferst	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 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 a 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.

4.7.1 Data Primitives

This section describes the implementation of the Data related primitives.

4.7.1.1 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->pMsdu, and may contain more than one byte even though the array is declared with a size of 1.

For MAC 2003 version the DataRequest structure is the following:

```
// 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 *pMsdu;     // Data will start at this byte
```

```
} mcpsDataReq_t;
```

For MAC 2006 version the DataRequest structure is the following:

```
typedef struct mcpsDataReq_tag {
  uint8_t  dstAddr[8];  // First 0/2/8 bytes is the address as defined by dstAddrMode
  uint8_t  dstPanId[2]; // 16 bit word converted to little endian byte array
  uint8_t  dstAddrMode;
  uint8_t  srcAddr[8];  // First 0/2/8 bytes is the address as defined by srcAddrMode
  uint8_t  srcPanId[2]; // 16 bit word converted to little endian byte array
  uint8_t  srcAddrMode;
  uint8_t  msduLength;  // 0-102
  uint8_t  msduHandle;
  uint8_t  txOptions;
  uint8_t  securityLevel;
  uint8_t  keyIdMode;
  uint8_t  keySource[8];
  uint8_t  keyIndex;
  uint8_t  *pMsdu;     //  Data will start at this address
} mcpsDataReq_t;
```

The security parameters have the same meaning as the ScanRequest security parameters.

4.7.1.2 Data Confirm

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

4.7.1.3 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->pMsdu, and may contain more than one byte even though the array is declared with a size of 1.

NOTE

Link Quality Indication (LQI) is used as part of the Data Indication structure representing an integer value from 0x00-0xFF where 0x00 equates to -100 dBm and the maximum value of 0xFF to -15 dBm.

For MAC 2006 version the DataIndication structure is the following:

```
// 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 *pMsdu;     // Data will start at this byte
 } mcpsDataInd_t;
```

For MAC 2006 version the DataIndication structure is the following:

```
typedef struct mcpsDataInd_tag {
  uint8_t  dstAddr[8];    // First 0/2/8 bytes is the address as defined by dstAddrMode
  uint8_t  dstPanId[2];   // 16 bit word converted to little endian byte array
  uint8_t  dstAddrMode;
  uint8_t  srcAddr[8];    // First 0/2/8 bytes is the address as defined by srcAddrMode
  uint8_t  srcPanId[2];   // 16 bit word converted to little endian byte array
  uint8_t  srcAddrMode;
  uint8_t  msduLength;    // 0-102 (101?)
  uint8_t  mpduLinkQuality;
  uint8_t dsn;
  uint32_t timeStamp;
  uint8_t securityLevel;
  uint8_t keyIdMode;
  uint8_t keySource[8];
  uint8_t keyIndex;
  uint8_t  *pMsdu;        // Data will start at this address inside the message.
} mcpsDataInd_t;
```

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

4.7.1.4 Poll Request

For MAC 2003 version the PollRequest structure is the following:

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

For MAC 2006 version the PollRequest structure is the following:

```
typedef struct mlmePollReq_tag {
  uint8_t  coordAddress[8];
  uint8_t  coordPanId[2];
  uint8_t  coordAddrMode;
  uint8_t  securityLevel;
  uint8_t  keyIdMode;
  uint8_t  keySource[8];
```

```
  uint8_t  keyIndex;
} mlmePollReq_t
```

The security parameters have the same meaning as the ScanRequest security parameters.

4.7.1.5 Poll Confirm

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

4.7.1.6 Poll Notify Indication

```
// Type: gNwkPollNotifyInd_c,
typedef struct nwkPollNotifyInd_tag {
  uint8_t  srcAddrMode;
         uint8_t  srcAddr[8];
         uint8_t  srcPanId[2];
} nwkPollNotifyInd_t;
```

4.7.1.7 Communications Status Indication

For MAC 2003 version the CommunicationStatusIndication structure is the following

```
// 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;
```

For MAC 2006 version the CommunicationStatusIndication structure is the following:

```
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;
  uint8_t  securityLevel;
  uint8_t  keyIdMode;
  uint8_t  keySource[8];
  uint8_t  keyIndex;
} nwkCommStatusInd_t;
```

- securityLevel — If the primitive was generated following a transmission instigated through a response primitive: The security level to be used. If the primitive was generated on receipt of a frame that generates an error in its security processing: The security level purportedly used by the received frame.
- keyIdMode — If the primitive was generated following a transmission instigated through a response primitive: The mode used to identify the key to be used. This parameter is ignored if the

securityLevel parameter is set to 0x00. If the primitive was generated on receipt of a frame that generates an error in its security processing: The mode used to identify the key purportedly used by the originator of the received frame. This parameter is invalid if the securityLevel parameter is set to 0x00.

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

4.7.2 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 add data to the pMsdu parameter.

```
nwkToMcpsMessage_t *mpPacket = MSG_Alloc(gMaxRxTxDataLength_c);
if(mpPacket != NULL)
{
        mpPacket->msgData.dataReq.pMsdu = "Message example";

        /* Data was available in the UART receive buffer. Now create an
        MCPS-Data Request message containing the UART data. */
        mpPacket->msgType = gMcpsDataReq_c;
        mpPacket->msgData.dataReq.msduLength = 16;

        /* Create the header using coordinator information gained during
        the scan procedure. Also use the short address we were assigned
        by the coordinator during association. */
        memcpy(mpPacket->msgData.dataReq.dstAddr, mCoordInfo.coordAddress, 8);
        memcpy(mpPacket->msgData.dataReq.srcAddr, maAddress, 2);
        memcpy(mpPacket->msgData.dataReq.dstPanId, mCoordInfo.coordPanId, 2);
        memcpy(mpPacket->msgData.dataReq.srcPanId, mCoordInfo.coordPanId, 2);
        mpPacket->msgData.dataReq.dstAddrMode = mCoordInfo.coordAddrMode;
        mpPacket->msgData.dataReq.srcAddrMode = gAddrModeShort_c;

        /* Request MAC level acknowledgement of the data packet */
        mpPacket->msgData.dataReq.txOptions = gTxOptsAck_c;

        /* Give the data packet a handle. The handle is
        returned in the MCPS-Data Confirm message. */
        mpPacket->msgData.dataReq.msduHandle = mMsduHandle++;

        /* Send the Data Request to the MCPS */
        (void) MSG_Send(NWK_MCPS, mpPacket);
}
```

An alternative way to provide the payload to MCPS-DATA.request is to include its contents in the data request message, like in the example below:

FLib_MemCpy(mpPacket->msgData.dataReq.pMsdu, "Message example", 16);

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<pMsg->msgData.dataInd.msduLength; i++)
      myBuffer[i] = pMsg->msgData.dataInd.pMsdu[i];
    break;
  case gMcpsDataCnf_c:
    // The MCPS-DATA.request has completed. Check status
    // parameter to see if the transmission was successful.
    break;
  case gMcpsPurgeCnf_c:
    // The MCPS-PURGE.request has completed.
    break;
  }
  MSG_Free(pMsg); // Free message ASAP.
  return gSuccess_c;
}
```

4.8 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 an 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.

4.8.1 Purge Primitives

This section describes the implementation of the Purge related primitives.

4.8.1.1 Purge Request

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

4.8.1.2 Purge Confirm

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

4.9 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 802.15.4 Standard.

4.9.1 RX Enable Request

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

4.9.2 RX Enable Confirm

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

4.10 Guaranteed Time Slots (GTS) Feature

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

4.10.1 GTS as a Device

It does not make sense for a device to allocate more than one Rx slot and one Tx slot (although it is possible to do so) because it is impossible for a device to differentiate two Tx slots of the same length. Analysis yields 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 802.15.4 Standard 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 similar. 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 MCU expensive and cannot be completed in IRQ context. The following steps describe the procedure for handling a GTS:

1. A beacon frame is received.
2. Time critical beacon frame processing is performed. This includes calculating various superframe timing parameters such as the expected end time of the CAP and the expected time of the next beacon frame arrival.
3. The GTS field of the beacon frame is 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.

4.10.2 GTS as PAN Coordinator

The PAN coordinator always accepts incoming GTS requests and it always allocates the requested GTS slot if the minimum length CAP 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 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 are calculated. This includes calculating the start and end times for all allocated GTS slots. The times are adjusted according to internal setup requirements and clock drift.
4. The gMlmeGtsAccess flag is cleared.
5. An MLME-GTS.indication message is generated (if applicable).

NOTE

The entire CFP of a superframe is skipped if the gMlmeGtsAccess is detected high at the beginning of the CFP. This is required 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) are deallocated immediately if the superframe configuration changes (macBeaconOrder or macSuperframeOrder are changed).

NOTE

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

GTS expiration is implemented according to the 802.15.4 Standard. The PAN coordinator deallocates stale slots automatically. The 802.15.4 Standard does not specify how to expire a GTS slot where data is sent unacknowledged. This implies that the PAN coordinator will not receive any acknowledgement frames.

The implementation in this case deallocates the GTS slot if no data has been transmitted in the slot for the specified number of superframes. For example, “counting” is based on Tx packets and not Rx acknowledgements.

4.10.3 Miscellaneous Items

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

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

As already stated, it is possible so skip a CFP due to GTS maintenance. Although this hazard exists, it should have minimal effects because GTS slots will more than likely rarely be allocated and deallocated except at feature setup and termination.

4.10.4 GTS Primitives

4.10.4.1 GTS Request

For MAC 2003 version the GtsRequest structure is the following:

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

For MAC 2006 version the GtsRequest structure is the following:

```
typedef struct mlmeGtsReq_tag {
  uint8_t  gtsCharacteristics;
  uint8_t  securityLevel;
  uint8_t  keyIdMode;
  uint8_t  keySource[8];
  uint8_t  keyIndex;
} mlmeGtsReq_t;
```

The security parameters have the same meaning as ScanRequest security parameters.

4.10.4.2 GTS Confirm

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

4.10.4.3 GTS Indication

For MAC 2003 version the GtsIndication structure is the following:

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

For MAC 2006 version the GtsIndication structure is the following:

```
typedef struct nwkGtsInd_tag {
  uint8_t  devAddress[2];
  uint8_t  gtsCharacteristics;
  uint8_t  securityLevel;
  uint8_t  keyIdMode;
  uint8_t  keySource[8];
  uint8_t  keyIndex;
} nwkGtsInd_t;
```

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

4.11 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, Freescale follows the ZigBee Security Services Specification V.092. So, the CCM security levels are used in place of the security suites described in the 802.15.4 Standard.

One current limitation in this implementation is that secured beacons are not 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 goes 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 4-3 shows the overhead added for each security level.

Table 4-3. Security Level Overhead

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

4.11.1 Security PIB Attributes

The ACL entries are statically allocated. The gNumAclEntryDescriptors may be defined in the `AppToMacPhyConfig.h` file. The default value for the number of ACL entries is 4.

It is possible to use the security PIB attributes as defined in the 802.15.4 Standard with the variation that the ACLSecurityMaterial always takes up the maximum (26 decimal) amount of bytes. The DefaultSecurityMaterial does the same, but it is always accessed directly.

It is also possible to write to the contents of the individual ACL entries using the Freescale specific security PIB attributes, see Table 4-1.

By setting the Freescale specific gMPibAclEntryCurrent_c security attribute to an ACL entry index between 0, and (gNumAclEntryDescriptors-1), 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.

For MAC 2006 the number of entries for *macKeyTable, KeyIdLookupList, KeyDeviceList, KeyUsageList, macDeviceTable* and *macSecurityLevelTable* are allocated statically. For each table the number of entries isdefined in `AppToMacPhyConfig.h` file: gNumKeyTableEntries_c, gNumKeyIdLookupEntries_c, gNumKeyDeviceListEntries_c, gNumKeyUsageListEntries_c, gNumDeviceTableEntries_c, gNumSecurityLevelTableEntries_c.

It is possible to read orwrite security PIB as defined in 802.15.4 2006 standard version. It is also possible to read or write the content of indiviual entry of each security PIB presented as a list.

4.11.2 Security Library

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

NOTE

These functions are not re-entrant, neither individually nor mutually. This requires all calls to these functions to happen from execution contexts that do not interleave, one of these being the execution context from which the MAC main function is called. The reason is that these functions modify some global variables without using a mutual exclusion mechanism.

4.11.2.1 Advanced Encryption Standard (AES)

4.11.2.1.1 Libraries

These libraries perform AES-128 cryptographic operation on a data set

- The 8-bit library is fully implemented in software and is optimized for the HCS08 platform. For all HCS08 projects, the architecture definitions must locate the internal buffers (MY_ZEROPAGE) in lower memory space for correct linking.
- The MC1322x ARM7 library makes use of the onboard Advanced Security Module (ASM).

4.11.2.1.2 Interface Assumptions

- All input/outputs are 16 bytes (128 bit).

NOTE

The function is not re-entrant. Also, it is of course not re-entrant with other functions calling this function (like SecLib_CcmStar).

- Users can point the ReturnData pointer to the same memory as either Data or Key if needed.

```
void SecLib_Aes
  (
  const uint8_t *pData, // IN:  Data to be en/de-crypted
  const uint8_t *pKey,  // IN:  128 bit key
  uint8_t *pReturnData  // OUT: Result (can be same address as
                        // pData or pKey)
  );
```

4.11.3 Counter with CBC-MAC (CCM*)

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

4.11.3.1 Interface Assumptions

Header, Message, and Integrity code have to be located in memory as they appear in the packet. That is, as a concatenated string in that order. For levels with encryption Message is encrypted in-place (overwriting Message).

Depending on the security level the function behaves as described in Table 4-4

Table 4-4. CCM* Internal Translation and Behavior for Different Security Levels

Level	Action	CCM* 'a' input	CCM* 'm' input
0	Nothing	N/A	N/A
1, 2, 3	Integrity only based on Header‖Message	Header‖Message	Empty
4	Encryption of Message only	Empty	Message

NOTE

The function is not re-entrant.

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

```
uint8_t SecLib_CcmStar
  (
  uint8_t * pHeader,            // IN/OUT: start of data to
                                // perform CCM-star on
  uint8_t headerLength,         // IN: Length of header field (a)
  uint8_t messageLength,        // IN: Length of data field (m)
  const uint8_t key[16],        // IN: 128 bit key
  const uint8_t nonce[13],      // IN: 802.15.4/Zigbee specific
                                //  nonce
  const uint8_t securityLevel, // IN: Security level 0-7
  gCcmDirection_t direction     // IN: Direction of CCM:
                                // gCcmEncode_c, gCcmDecode_c
  );
```

Chapter 5
APP/ASP Layer Interface Description

This section describes the Application (APP)/Application Support Package (ASP) interface. As described earlier in this manual, the user must be aware that not all functions and primitives are available on both supported platforms, i.e., some are specific to a single platform and are noted as such.

5.1 General APP/ASP Interface Information

The interface between the APP and the ASP is based on direct function calls. Seventeen functions are implemented in the ASP part of the MAC. The application layer can use these functions after including the `AppAspInterface.h` header file. The ASP functions prototypes are highlighted in this section. However, refer to Section 5.3, “APP to ASP Interface” for a more complete description.

The interface between the ASP and the APP is based on service primitives passed from one layer to the other through a layer Service Access Point (SAP).

5.1.1 uint8_t ASP_APP_SapHandler(aspToAppMsg_t *pMsg)

The ASP to APP SAP ASP_APP_SapHandler() passes primitives from the ASP to the APP. The ASP_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 MSG_Send(ASP_APP, msg) macro. ASP to APP service primitives use the same type of messages as defined in the `AppAspInterface.h` interface header file. The macros are defined in the `MsgSystem.h` header file.

Because the ASP to APP interface is based on messages being passed to a SAP, each message needs to have an identifier. These identifiers are shown in the enumeration in Table 5-1.

Table 5-1. 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

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.

5.2 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 lists as shown in Table 2-1, Table 2-2, Table 2-3, and Table 2-4 for implemented primitives. All indication primitives are sent in messages that must be freed by MSG_Free() as described in Section 2.4.6, “MSG_Free”.

NOTE

Some primitives are not available for the AMR7 based MC1322x platform.

5.2.1 Wake Indication

Available only on HCS08 based platforms, the ASP-WAKE.Indication primitive is sent to the APP when the transceiver 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 transceiver in doze mode again after the wake indication has been processed by the ASP_APP SAP. Thus, the APP has the opportunity 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;
```

5.2.2 Idle Indication

Available only on HCS08 based platforms, 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 super frame’s idle portion. The `timeRemaining` parameter is the number of CAP symbols left. 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;
```

5.2.3 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 super frame’s inactive portion. 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;
```

5.2.4 Event Indication

Available only on HCS08 based platforms, 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;
```

5.2.5 ASP to APP Message Union

The aspToAppMsg_t structure union to passes ASP specific messages from application layer to the ASP layer.

```
// ASP to application message
typedef struct aspToAppMsg_tag {
  uint8_t msgType;
  union {
    appWakeInd_t            appWakeInd;
    appIdleInd_t            appIdleInd;
    appInactiveInd_t        appInactiveInd;
    appEventInd_t           appEventInd;
  } msgData;
} aspToAppMsg_t;
```

5.2.6 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, so they do not end up in the APP SAP.

Example 5-1. 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 transceiver
 // is re-entering doze mode.
```

```
      DoSomeThingWhileAwake();
      // When returning from here, transceiver enters doze mode asap.
}
    else {
      // transceiver 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;
}
```

5.3 APP to ASP Interface

Functions used by the APP layer for requesting different ASP functionality are common to all platforms with the following exceptions:

- The following functions are not available for the ARM7 based MC1322x platform:
  - Asp_GetInactiveTimeReq
  - Asp_DozeReq
  - Asp_AutoDozeReq
  - Asp_AcomaReq
  - Asp_HibernateReq
  - Asp_EventReq
  - Asp_SetNotifyReq
  - Asp_SetMinDozeTimeReq
  - Asp_WakeReq
  - Asp_PortReq
  - Asp_DdrReq
  - Asp_ClkoReq
- The following functions are not available for the HCS08 platform:
  - Asp_SetDemodulatorType
  - Asp_EnableComplementaryPAOutput
  - Asp_ConfigureRFCtlSignals
  - Asp_SetPowerLevelLockMode

5.3.1 Get MAC Time Functions

Functions are available to acquire MAC internal time values. The MC1319x, MC1320x, and MC1321x devices have a separate transceiver with onboard timers. In contrast ,the MC1322x devices provide a timer in the onboard MAC Accelerator block (MACA).

5.3.1.1 void Asp_GetTimeReq(zbClock24_t *time)

Available only on HCS08 based platforms, this function requests the tranceiver's current internal event timer value.

The time pointer points to a 3-byte array where the transceiver's internal 24 bit event timer value will be copied. The internal event timer's current value (0x000000 to 0xFFFFFF) is little endian. The function does not return any value because the call is always successful.

5.3.1.2 void Asp_GetTimeReq(zbClock32_t *time)

Available only on MC1322x family, this function requests the MACA current internal event timer value.

The time pointer points to a 4-byte array where a 30-bit value represents the MACA symbol clock time. This value is obtained by reading the MACA 32-bit Clock Register (MACA_CLK) (which runs at the 250KHz bit clock) and shifting the register value right 2 bits. The value is little endian. The function does not return any value because the call is always successful.

5.3.2 uint8_t Asp_GetInactiveTimeReq(zbClock24_t *time)

Available only on HCS08 based platforms, this function requests the remaining time in the super frame's inactive portion (related to beacon mode).

The time pointer points to a 3-byte array where the superframe's inactive portion 24-bit time remaining will be copied. The remaining time in the superframe's inactive portion (0x000000 to 0xFFFFFF) is little endian. The function will only return gSuccess_c when called during the inactive portion of a super frame, and the transceiver is not in Doze, or Hibernate mode. Otherwise, the returned value will be gInvalidParameter_c.

5.3.3 uint8_t Asp_DozeReq(zbClock24_t *dozeDuration, uint8_t clko_en)

Available only on the HCS08 platforms, this function requests a transceiver shut down for a given amount of time in symbols.

The dozeDuration pointer points to a 3-byte array which contains the maximum time in number of symbols that the transceiver will be in Doze mode. The transceiver can be awakened prematurely from Doze Mode by a signal on the ATTN pin. CLKO is automatically started again when the transceiver leaves Doze Mode.

The CLKO output pin stops providing a clock signal to the CPU while dozing if clko_en is 0 (FALSE). The CLKO output pin continues to provide a clock signal while dozing if clko_en is 1 (TRUE).

If Doze mode is not possible because the transceiver is busy, the function returns the gInvalidParameter_c. Otherwise, the function returns gSuccess_c.

The dozeDuration parameter is modified by the function. So, if the transceiver is idle, the requested doze duration is granted and the DozeDuration parameter is the same as the time requested. However, if the transceiver is in a timed wait state with a duration shorter than the requested duration, the transceiver will doze until the wait state completes. The DozeDuration parameter will be the difference between the requested doze duration and the end time of the transceiver wait state.

5.3.4 uint8_t Asp_AutoDozeReq(bool_t autoEnable, bool_t enableWakeIndication, zbClock24_t *autoDozeInterval, uint8_t clko_en)

Available only on the HCS08 platforms, this function requests an automatic shut down of the transceiver during idle periods.

The CLKO output pin stops providing a clock signal to the CPU while dozing if clko_en is 0 (FALSE). The CLKO output pin continues to provide a clock signal while dozing if clko_en is 1 (TRUE).

The autoDozeInterval pointer points to a 3 byte array which contains the suggested period in symbols in which the transceiver will be in Doze mode. This interval may be overridden if Doze mode is interrupted by an external signal (ATTNBi pin).

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. Freescale recommends using the ASP-WAKEIndication for simple processing during auto doze because it will occur frequently (if enabled) and the auto doze feature is blocked during the processing of the indication in the ASP_APP SAP. The function always returns gSuccess_c.

5.3.5 uint8_t Asp_AcomaReq(uint8_t clko_en)

Available only on the HCS08 platforms, this function requests a transceiver shut down. The CLKO output pin stops providing a clock signal to the CPU if clko_en is 0 (FALSE). The CLKO output pin continues to provide a clock signal if clko_en is 1 (TRUE). Only a signal on the ATTNBi pin of the transceiver or a power loss can bring the transceiver out of Acoma mode. CLKO is automatically started again when transceiver leaves Acoma mode.

The Acoma mode is not suited for beaconed operation and Doze mode should be used instead when transceiver internal timers are required. One primary difference between Acoma mode and Hibernate mode is that CLKO can be generated during Acoma mode, which is not possible in Hibernate. Acoma mode does not support timer wakeup, which is possible during Doze mode.

The function returns gSuccess_c if the transceiver is in Idle mode. Otherwise the return value is gInvalidParameter_c.

5.3.6 uint8_t Asp_HibernateReq(void)

Available only on the HCS08 platforms, the hibernate request shuts down the transceiver.

The CLKO output pin stops providing a clock signal to the CPU. Only a signal on the ATTNBi pin of the transceiver or a power loss can bring the transceiver out of Hibernate mode. CLKO is automatically started again when transceiver leaves Hibernate mode. The Hibernate mode is not adequate for beaconed operation. Doze mode should be used instead when internal transceiver timers are required.

The function returns gSuccess_c if the transceiver is in Idle mode. Otherwise, the return value is gInvalidParameter_c.

5.3.7 uint8_t Asp_EventReq(zbClock24_t *time)

Available only on the HCS08 platforms, this function can be used to schedule a notification for an application event. The notification is a single instance event. If there is any conflict with the MAC operation a gInvalidParameter_c value is returned, otherwise the function returns gSuccess_c.

The time parameter is pointer to a 3-byte little endian integer symbol time. The event time is relative to moment when the function was called.

5.3.8 Device Reference Oscillator Trim Functions

The MC1319x, MC1320x, and MC1321x devices use a 16 Mhz transceiver reference crystal oscillator and these devices provide onboard trimmable capacitive loading in addition to external load capacitors to the crystal. In contrast ,the MC1322x devices provide complete onboard load capacitance to a 24MHz device reference crystal oscillator. A trim function is provided for each platform as described in the following sections.

5.3.8.1 void Asp_TrimReq(uint8_t trimValue)

Available only on HCS08 based platforms, this function sets the trim capacitor value for the transceiver 16MHz reference oscillator. Upon call, the trim capacitor value contained in the 8-bit parameter is programmed to the transceiver.

The function does not return any value because the call is always successful.

5.3.8.2 void Asp_TrimReq(uint8_t fineTune, uint8_t coarseTune)

Available only on MC1322x based platforms, this function trims the device 24MHz reference oscillator (via changing the load capacitance) through programming the CRM register XTAL Control (XTAL_CNTL)

The function does not return any value because the call is always successful.

5.3.9 uint8_t Asp_SetNotifyReq(uint8_t notifications)

Available only on the HCS08 platforms, this function 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 5.2.2, "Idle Indication") is sent.
3. gAspNotifyInactive_c — ASP-INACTIVE Indication (See Section 5.2.3, "Inactive Indication") 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. If beacons are part of the MAC feature the value returned is always gSuccess_c. Otherwise the return value is gInvalidParameter_c.

5.3.10 uint8_t Asp_SetMinDozeTimeReq(zbClock24_t *minDozeTime)

Available only on the HCS08 platforms, this function sets the default minimum transceiver doze time. If the MAC cannot doze for at least the minimum doze time, then it will not enter Doze mode. For example, if the doze request is issued 3ms before the end of a beacon period, the MAC will not enter Doze mode since the default minimum doze time is 4ms. However, if the minimum doze time is changed to 2ms, then the MAC will doze for 2ms, and wake up 1ms before the next beacon. Assume the same timing in both examples. The function always returns gSuccess_c value.

5.3.11 void Asp_TelecTest(uint8_t mode)

This function executes one of the TELEC test sequences. After finishing the test, the device should either be reset using MLME-RESET.request, or power cycle.

The mode parameter of the request can be one of the following as shown in Table 5-2.

Table 5-2. TELEC Mode Parameter

Mode	Description
gTestForceIdle_c	Stop the test currently running.
gTestPulseTxPrbs9_c	Continuously transmit a PRBS9 pattern.
gTestContinuousRx_c	Sets the device into continuous RX mode.
gTestContinuousTxMod_c	Sets the device to continuously transmit a 10101010 pattern.
gTestContinuousTxNoMod_c	Sets the device to continuously transmit an un-modulated carrier wave.

The function does not return any value because the call is always successful.

5.3.12 Asp_TelecSetFreq(uint8_t channel)

This function sets the logical channel used when running the TELEC test sequences. The logicalChannel parameter can be any value between 0x0B, and 0x1A. The function does not return any value because the call is always successful.

5.3.13 Functions for Setting RF TX Power Level

The functions for setting device transmit power level are affected by the target platform:

- HCS08 based platforms use only “Asp_SetPowerLevel(uint8_t powerLevel)”
- MC1322x based platforms use:
  - “Asp_SetPowerLevel(uint8_t powerLevel)” to set the power level
  - “Asp_SetPowerLevelLockMode(bool_t enableLock)” to restrict the power levels for use with external amplification

5.3.13.1 HCS08 Based Platforms (Asp_SetPowerLevel(uint8_t powerLevel))

The “Asp_SetPowerLevel(uint8_t powerLevel)” for HCS08 based platforms allows 16 possible values for the powerLevel parameter, i.e., hex 0x00 to 0x0E. The relationship between the powerLevel parameter and the actual transmit power is not perfectly linear or exponential. Table 5-3 shows the typical transmit power level, and the TX level applies to either single-port or dual-port PA mode in the MC1320x and MC1321x platforms.

Table 5-3. HCS08 PA Level vs. Output Power

Power Level (Hex)	Transmit Power (dBm)
0	-16.6
1	-16
2	-15.3
3	-14.8
4	-8.8
5	-8.1
6	-7.5
7	-6.9
8	-1
9	-0.5
A	0
B	0.4 (default)
C	2.1

Table 5-3. HCS08 PA Level vs. Output Power (continued)

Power Level (Hex)	Transmit Power (dBm)
D	2.8
E	3.5

5.3.13.2 MC1322x Based Platform (Asp_SetPowerLevel(uint8_t powerLevel) and Asp_SetPowerLevelLockMode(bool_t enableLock))

For MC1322x platforms the “Asp_SetPowerLevel(uint8_t powerLevel)” function has a wider range of values, i.e., from (hex) 0x00 to 0x11. Table 5-4 shows typical TX output power vs. programmed value. The listed output power is measured at the RF_RX_TX port of the device.

The use of the “Asp_SetPowerLevel(uint8_t powerLevel)” function and the MC1322x transmit power level settings are affected by the use of external amplification (an external hardware power amplifier (PA)). When an external PA is used:

— The highest 802.15.4 channel (Channel 26) must be disabled, and its use is disallowed

— The allowable power levels are restricted.

— The user must enable the restriction of the TX power levels (called powerlock) via the “Asp_SetPowerLevelLockMode(bool_t enableLock)” function

Available only on the MC1322x platform, the “Asp_SetPowerLevelLockMode(bool_t enableLock)” function is used to enable or disable the power level lock mode. When the power level lock mode is applied, the output power level is restricted to the following values (hex): 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x0C. The Asp_SetPowerLevel function will return gDenied_c if a different power level is requested when the lock mode is applied. Also in this mode, the Channel 26 cannot be selected through MACPib.Set primitive (returned status = gDenied_c).

The enableLock parameter selects the current power level lock mode on the following basis:

- enableLock = TRUE (1) - MC1322x Power Level lock mode is enabled
- enableLock = FALSE (0) - MC1322x Power Level lock mode is disabled

Based on the current selected power level and communication channel selected at the function call moment, the Asp_SetPowerLevelLockMode can return the following values:

- gSuccess_c = The requested power lock mode was applied.
- gDenied_c = The power level lock mode could not be enabled. The current power level is not allowed in this mode, or the channel 26 is currently selected.

Table 5-4. MC1322x PA Level vs. Output Power

Power Level (Hex)	Transmit Power (dBm)	Available for PowerLock[1]
0	-30	**Yes**
1	-28	**Yes**

Table 5-4. MC1322x PA Level vs. Output Power (continued)

Power Level (Hex)	Transmit Power (dBm)	Available for PowerLock[1]
2	-27	**Yes**
3	-26	**Yes**
4	-24	**Yes**
5	-21	**Yes**
6	-19	**Yes**
7	-17	**Yes**
8	-16	No
9	-15	No
A	-11	No
B	-10	No
C	-4.5	**Yes**
D	-3	No
E	-1.5	No
F	-1	No
10	1.7	No
11	3	No

[1] When Power Lock is enabled only the power settings shown as available may be used. This feature is intended for use with an external PA.

5.3.14 uint8_t Asp_GetPowerLevel(void)

This function returns the current programmed power level of the transceiver. This function returns the value set by the user through the call to:

Asp_SetPowerLevel(uint8_t requestedPowerLevel)

If no call has been previously made, the default start value will be returned - gAspPowerLevel_0dBm_c.

5.3.15 void Asp_SetDemodulatorType(bool_t demDCDenable)

Available only on the MC1322x platform, this function selects either the Differentially-coherent Chip Detection (DCD) or the Non-coherent Chip Detection (NCD) mode demodulation in the MC1322x receiver. The DCD mode is more robust, but has less sensitivity; it is also the default mode.

The demDCDenable parameter selects the actual demodulator type on the following basis:

- demDCDenable = TRUE (1) - DCD demodulation is enabled (default)
- demDCDenable = FALSE (0) - NCD demodulation is enabled.

5.3.16 void Asp_EnableComplementaryPAOutput(bool_t enable)

The MC1322x device has two transmit modes:

- A primary mode where a shared single-ended port (RF_RX_TX) is used for both receive and transmit.
- A secondary mode where a secondary dual-port PA with complementary outputs (ports PA_POS and PA_NEG) is used for transmit. In this mode the RF_RX_TX port is used only for receive.

To control use of the transmit PAs (only available on the MC1322x platform), the "Asp_EnableComplementaryPAOutput(bool_t enable)" function is used to enable or disable the complementary PA outputs. The default state of the complementary PA outputs is determined by the BeeKit via the "MC1322x User Defined Target Editor" tool.

The enable parameter selects the current PA output path on the following basis:

- enable = TRUE (1) - MC1322x Complementary PA Output is enabled
- enable = FALSE (0) - MC1322x Complementary PA Output is disabled

5.3.17 uint8_t Asp_ConfigureRFCtlSignals(AspRfSignalType_t signalType, AspRfSignalFunction_t function, bool_t gpioOutput, bool_t gpioOutputHigh)

The MC1322x has four I/O signals that can optionally be used to control external RF hardware: ANT_1 (GPIO42), ANT_2 (GPIO43), TX_ON (GPIO44), RX_ON (GPIO43). The advantage of using these signals is that they can be controlled by the radio hardware without need of real-time software support.

Available only on the MC1322x platform, the "uint8_t Asp_ConfigureRFCtlSignals(AspRfSignalType_t signalType, AspRfSignalFunction_t function, bool_t gpioOutput, bool_t gpioOutputHigh)" function is used to configure these RF control signals.

The function can configure one of the above signals at a time, with the possibility of selecting between Automatic RF Control (Function 1 or Function 2) or general purpose I/O (GPIO).

NOTE

Refer to the *MC1322x Reference Manual,* for use and operation of RF circuitry and these control signals.

Provided values for Asp_ConfigureRFCtlSignals signalType:

```
typedef enum {
  gAspRfSignalANT1_c,
  gAspRfSignalANT2_c,
  gAspRfSignalTXON_c,
  gAspRfSignalRXON_c,
  gAspRfSignalMax_c
}AspRfSignalType_t;
```

Provided values for Asp_ConfigureRFCtlSignals signalFunction:

```
typedef enum {
  gAspRfSignalFunctionGPIO_c,
  gAspRfSignalFunction1_c,
```

```
  gAspRfSignalFunction2_c,
  gAspRfSignalFunctionMax_c
}AspRfSignalFunction_t;
```

The Table 5-5 shows the possible values for the RF control signal parameters.

Table 5-5. Asp_ConfigureRFCtlSignals Function Parameters

signalType	function	gpioOutput	gpioOutputHigh
ANT_1 (GPIO42)	FunctionGPIO (Rf Ctl disabled)	FALSE (In)	Ignored
	FunctionGPIO (Rf Ctl disabled)	TRUE (Out)	TRUE(high)/FALSE(low)
	Function1(Rf controlled)	Ignored	Ignored
ANT_2 (GPIO43)	FunctionGPIO (Rf Ctl disabled)	FALSE (In)	Ignored
	FunctionGPIO (Rf Ctl disabled)	TRUE (Out)	TRUE(high)/FALSE(low)
	Function1 (Rf controlled)	Ignored	Ignored
TX_ON (GPIO44)	FunctionGPIO (Rf Ctl disabled)	FALSE (In)	Ignored
	FunctionGPIO (Rf Ctl disabled)	TRUE (Out)	TRUE(high)/FALSE(low)
	Function1 (Rf controlled)	Ignored	Ignored
	Function2 (Rf controlled)	Ignored	Ignored
RX_ON (GPIO45)	FunctionGPIO (Rf Ctl disabled)	FALSE (In)	Ignored
	FunctionGPIO (Rf Ctl disabled)	TRUE (Out)	TRUE(high)/FALSE(low)
	Function1 (Rf controlled)	Ignored	Ignored
	Function2 (Rf controlled)	Ignored	Ignored

This function can return the following values:

- gSuccess_c = Function parameters are valid and the requested configuration was applied.
- gInvalidParameter_c = Function parameters are invalid and the requested configuration was denied.

5.3.18 uint8_t Asp_GetMacStateReq(void)

Get basic state of the MAC. The caller can use this information to determine if it is safe to go into one of the deep sleep modes! If the MEM, SEQ, and MLME state machines are not in idle state. it returns gAspMacStateBusy_c. If the queues are empty the functions returns gAspMacStateNotEmpty_c. In all other cases the returned value is gAspMacStateIdle_c.

5.3.19 void Asp_WakeReq(void)

Available only on HCS08 based platforms, this function wakes up the transceiver from Doze or Hibernate mode.

- The ATTNBi pin of the transceiver must be wired to an MCU port pin for this primitive to function. Otherwise, it has no effect.
- The MC1319xDrv_AttEnable, and MC1319xDrv_AttDisable macros must be defined in the `MC1319xDrv.c` file.
- The MC1319xDrv_AttEnable must set the MCU port pin to logic high
- The MC1319xDrv_AttDisable must set it to logic low.

The function does not return any value because the call is always successful.

5.3.20 HCS08 Platform Transceiver GPIO Functions

On the HCS08 based platforms, the transceiver is a separate device and has an independent 7-bit GPIO port (designated GPIO1 - GPIO7) from the MCU. These functions relate to the transceiver GPIO port.

NOTE

GPIO1 and GPIO2 have alternate hardware functions that are used by the 802.15.4 MAC software. As such, they are not available for user application purposes and the following functions apply only to GPIO3 - GPIO7

5.3.20.1 void Asp_PortReq(uint8_t portWrite, uint8_t portValue, uint8_t *CnfPortResult)

Available only on HCS08 based platforms, this function reads or writes the transceiver 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).
- If FALSE, the GPIO data register (only bits 3-7) will be copied at the address given by CnfPortResult.

The function does not return any value because the call is always successful.

5.3.20.2 void Asp_DdrReq(uint8_t directionMask)

Available only on HCS08 based platforms, this function 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 2:0 of the mask are ignored.

The function does not return any value because the call is always successful.

5.3.21 uint8_t Asp_ClkoReq(bool_t clkoEnable, uint8_t clkoRate)

Available only on HCS08 based platforms, this function sets and/or enables the transceiver CLKO output (commonly used as an external clock source to the HCS08).

- If clkoEnable is TRUE, CLKO is made active, otherwise it is disabled.
- The CLKO output frequency is programmed depending on the value contained in clkoRate per the CLKO frequency selection of the transceiver. The clkoRate can be assigned the values shown in Table 5-6.

Table 5-6. CLKO Values

clkoRate	CLKO frequency
0	16 MHz
1	8 MHz
2	4 MHz
3	2 MHz
4	1 MHz
5	62.5 KHz
6	31.738 KHz(default)
7	16.393 KHz

5.3.22 Examples of APP to ASP calls

This section provides two 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 5-2. Getting the Current Transceiver Clock

```
// APP layer must allocate a 3 bytes buffer, where the clock value
// shall be retrieved by the function
int8_t currentTime[3];

//call the ASP function
Asp_GetTimeReq(currentTime);
// Now currentTime contains the value of the transceiver clock
```

Example 5-3. Start Auto Doze

```
// APP layer must allocate a buffer for the requested doze time
uint8_t autoDozeInterval[3];
bool_t weAreAutoDozing = FALSE;

// 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.

//autoEnable parameter will be set to TRUE;
//enableWakeIndication parameter will be set to TRUE;
```

```
// Values greater than 1 byte must be little endian byte arrays.
autoDozeInterval[0] = 0xB4;
autoDozeInterval[1] = 0x4C;
autoDozeInterval[2] = 0x04;
// Call the ASP function

if (Asp_DozeReq(TRUE, TRUE, autoDozeInterval, FALSE)==gSucces_c)
{
weAreAutoDozing = TRUE;
}
```