802.15.4 MAC/PHY Software

User's Guide

802154MPSUG/D
Rev. 0.0, 11/2004

Launched by Motorola
freescale™
semiconductor

How to Reach Us:

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

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

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

Home Page:
www.freescale.com

Contents

About This Book

This user's guide describes how to use the Freescale 802.15.4 MAC/PHY software. It describes how to access and use the Medium Access Control/Physical Layer (MAC/PHY) and provides detailed programming descriptions which will allow users to develop upper layer or application code for this product.

This manual is best used together with the *802.15.4 MAC/PHY Software Reference Manual*, 802154MPSRM/D, so that any detailed and specific questions can be answered while reading this manual.

The Freescale 802.15.4 software is designed for use with the Freescale MC13192 802.15.4 transceiver as described in the *MC13192/93 Data Sheet*, MC13192DS/D. The software package, the MC13192, and the HC08 MCU, form the Freescale 802.15.4 platform solution.

Organization

This document is organized into three chapters:

Chapter 1	**Overview** — This chapter presents a brief overview of the Freescale 802.15.4 software.
Chapter 2	**Interfacing to the 802.15.4 MAC Software** — This chapter describes how to interface a user application to the MAC and how to use the MAC interface functions.
Chapter 3	**Creating an Application** — This chapter guides users through the steps required to create a basic, non-beacon network with one coordinator and one device.

Conventions

This document uses the following notational conventions:

- Courier monospaced type indicates commands, command parameters, code examples, expressions, data types, and directives e.g. `ret = MSG_Send(NWK_MLME, msg);`
- All source code examples are written in C.

Definitions, Acronyms, and Abbreviations

The following list defines the abbreviations used in this document.

API	Application Programming Interface
ASP	Application Support Package
CAP	Contention Access Period
CSMA-CA	Carrier Sense Multiple Access with Collision Avoidance
ED	Energy Detection
FFD	Full Function Device
GTS	Guaranteed Time Slot
ISR	Interrupt Service Routine
LQI	Link Quality Indication
MAC	Medium Access Control
MCPS	MAC Common Part Sublayer
MCU	Micro Control Unit
MLME	MAC subLayer Management Entity
NWK	Network
PAN	Personal Area Network
PANID	PAN Identification
PHY	PHYsical Layer
PIB	PAN Information Base
RAM	Random Access Memory
RFD	Reduced Function Device
SAP	Service Access Point
SW	Software
WPAN	Wireless Personal Area Network

References

The following documents are referenced in this document.

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

[2] *MC13192 Data Sheet*, MC13192DS/D, Freescale Semiconductor, 2004

[3] *802.15.4 MAC/PHY Software Reference Manual*, 802154MPSUM/D, Freescale Semiconductors, 2004

[4] *HCS08 Family Reference Manual*, Motorola Inc., 2003

Revision History

The following table summarizes revisions to this manual since the previous release (Rev. 0.0).

Revision History

Location	Revision
Entire Document	New document. 0.0

Chapter 1
Overview

This chapter presents a brief overview of the Freescale 802.15.4 software. In Section 1.1, the main concept of the IEEE 802.15.4 standard is explained. Section 1.2 provides a brief introduction to the full function device and the reduced function device and the use cases associated with each one. Section 1.3 provides a system overview. In this section, pay special attention to the concepts presented regarding the three interfaces present in this implementation:

1. MLME
2. MCPS
3. ASP

These three interfaces comprise the Application Programming Interface (API) and are described in detail in Chapter 2. Chapter 3 guides users through the making of an application using the IEEE 802.15.4 software and the API as described in Chapter 2.

1.1 IEEE 802.15.4 Background

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

- An over the air data rate of 250 kbit/s in the 2.4GHz band.
- 16 independent communication channels in the 2.4GHz 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.
- Devices inform the application about the quality of the wireless link - Link Quality Indication (LQI).

The IEEE 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:

The star network topology In a star network all communication in the network is either to the PAN coordinator or from the PAN coordinator. That is, communication between non-PAN coordinator devices is not possible.

The 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.

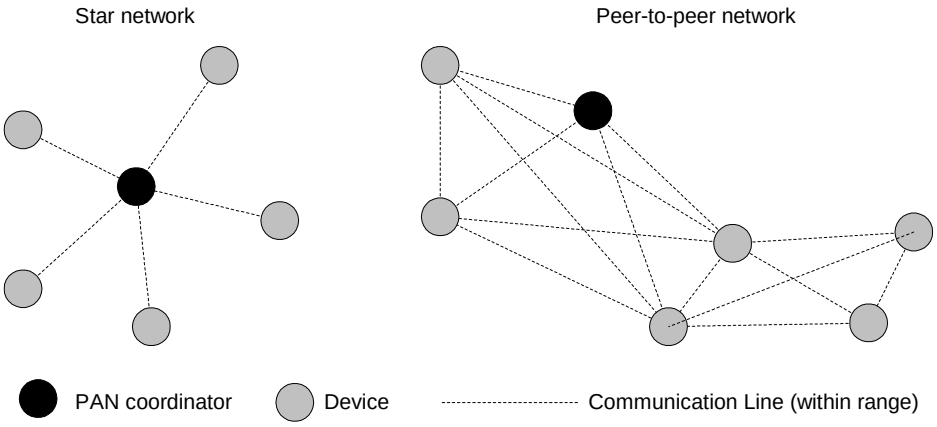

Figure 1. Peer-to- Peer and Star Network (No PAN Coordinator)

If a device wants to join an IEEE 802.15.4 network it will have to associate with a device that is already part of the network. That allows other devices to associate with it. A device can only be associated with one other device but multiple devices can be associated with the same device as shown in Figure 1. 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 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.

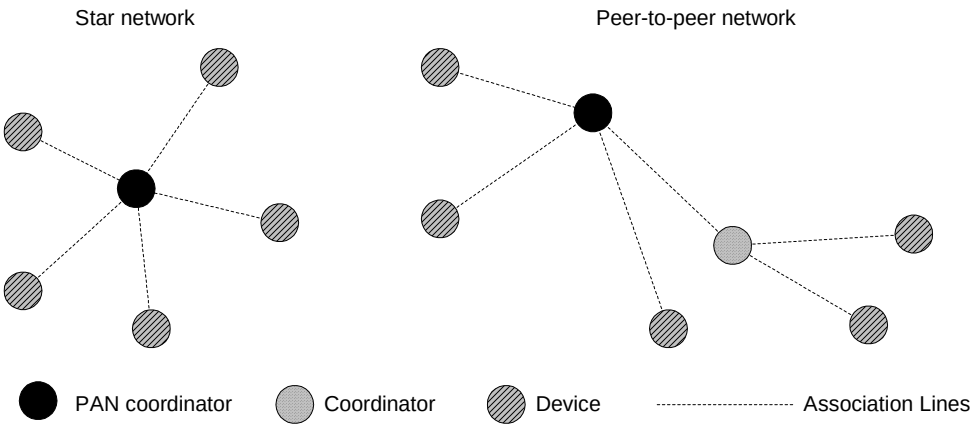

Figure 2. Peer and Star Network (With PAN Coordinator)

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 its 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

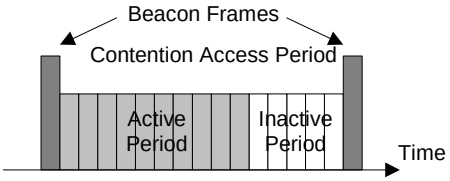

Figure 3. Beacon Frame Timing

In non-beacon mode data transmissions data transmission can take place at any time. For both non-beacon and beacon networks, the application can choose to transmit data in two ways.

- **Direct data transfer -** Data exchanged between a device and a coordinator in a non-beacon network using direct data transfer takes place as soon as the channel is free using CSMA-CA.
- **Indirect data transfer -** Data from a device to a coordinator in a beacon network using indirect data transfer is sent in the CAP as soon as the channel is free using CSMA-CA.

1.2 Device Types

The basic device types for the Freescale 802.15.4 software are as follows:

- **Reduced Function Device (RFD) -** This device contains a subset of the IEEE 802.15.4 features. The RFD can only act as a device in a network. Normally the RFD would be used for battery powered devices, since a device with a pure device capability is very power efficient. Examples of RFDs could be light switches and temperature sensors.
- **Full Function Device (FFD) -** This device contains all of the IEEE 802.15.4 features. The FFD can act as device or coordinator in a network. The FFD is targeted to be used for backbone powered devices. An FFD that take on the role as coordinator uses significantly more power than an RFD in the device role and is more complex and therefore requires more code and memory space than the RFD. Examples of FFDs could be light controllers with router capability (light bulbs) and PAN coordinators (main network controller).

To allow users to select the best memory solution, Freescale provides both the RFD and the FFD as precompiled libraries. In addition, Freescale provides a small number of derivatives of both the FFD and the RFD devices to further reduce memory requirements. For example, if users want to implement an FFD device that does not require beacon capability, they can save almost 9 kb of code space by selecting the FFD library that does not contain this feature. Table 1 shows the MAC device types as provided by Freescale.

Table 1. MAC Device Types

Device type	Description	Code size
FFD	Full-blown FFD. Contains all 802.15.4 features including security.	37k
FFDNGTS	Same as FFD but no GTS capability.	33k
FFDNB	Same as FFD but no beacon capability.	28k
FFDNBNS	Same as FFD but no beacon and no security capability.	21k
RFD	Reduced function device. Contains 802.15.4 RFD features.	29k
RFDNB	Same as RFD but no beacon capability.	25k
RFDNBNS	Same as RFD but no beacon and no security capability.	18k

Table 1 shows that a significant code reduction is achieved by proper device type library selection.

NOTE

Most low cost, low power applications fit into a standalone embedded device model where the application and MAC/PHY reside on the same processor.

1.3 System Overview

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

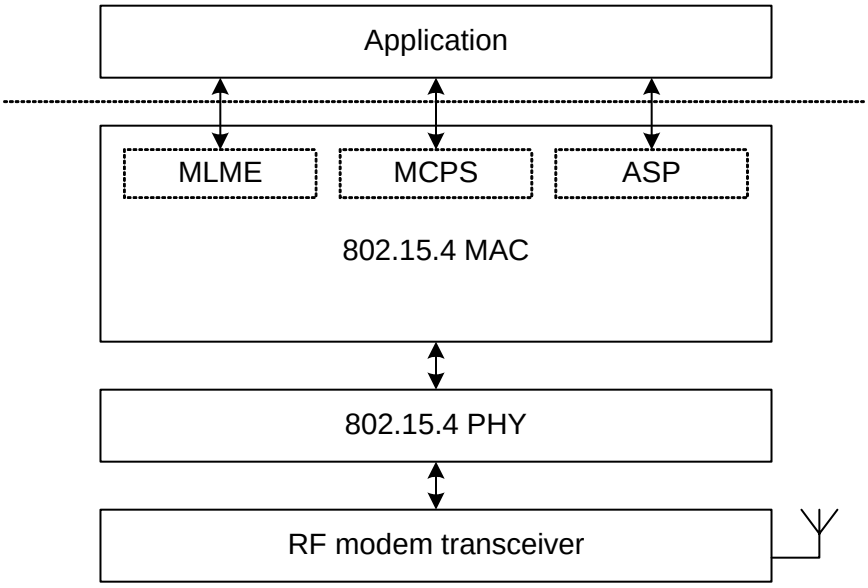

Figure 4. System Block Diagram

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 4, is the 802.15.4 MAC (or just MAC). The MAC provides three interfaces to the application.

1. **MLME 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 IEEE 802.15.4 specification.

2. **MCPS 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 IEEE 802.15.4 specification.

3. **ASP interface -** This interface is used for various application support features. For example, the application can request that the hardware must enter a low power mode. This interface is proprietary to Freescale.

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

NOTE

The application must use the three MAC interfaces to implement the desired functionality. Chapter 3 describes in detail how to implement a simple application. Chapter 2 describes the interfaces in complete detail.

Chapter 2
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 the example application described in Chapter 3 for a detailed walk-through of how to create a more advanced application. For a more detailed description of all MAC primitives refer to the *802.15.4 MAC/PHY Software Reference Manual*, 802154MPSRM/D. Freescale recommends having the reference manual at hand when reding this guide because subtle details (for reasons of readability) are not included in this guide. A brief overview of the MAC interfaces are given in Section 2.3 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.

2.1 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 the *802.15.4 MAC/PHY Software Reference Manual*, 802154MPSRM/D. Figure 5 shows the three interfaces to the MAC.

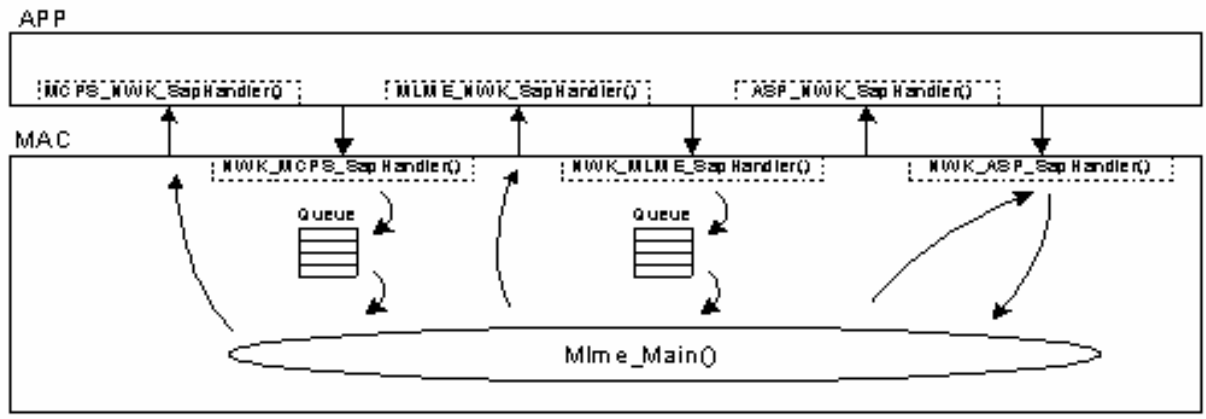

Figure 5. MAC Interfaces

Messages are sent to a Service Access Point (SAP) function which is responsible for handling the message. Six SAPs exists, three for the communication stream to the MAC and three for the communication to the application layer.
The following list shows the three 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 2.5 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 2.5 for a detailed description.

3. `APP_ASP_SapHandler()` - The Application Support Package (ASP) interface is provided in order to support Freescale specific functionality such as enabling the low-power modes of the MAC. This SAP is used for all ASP request from the application layer to the MAC. See Section 2.6 for a detailed description.

The following list shows the three Service Access Points 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 2.5 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 2.5 for a detailed description.

3. `ASP_APP_SapHandler()`- Used for receiving all ASP indications from the MAC. See Section 2.6 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.
Parts of the MAC are executed through a synchronous, interrupt driven part. This part is referred to as the MC13192 ISR because the controlling interrupts are generated by the MC13192 radio. MC13192 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 this interrupt. The following timing constraint must be kept by the application in order to meet MAC interrupt latency demands:

- Within a peiod of 64 µs the application must disable the MC13192 interrupts for a duration of *maximum 10 µs*.

2.2 Include Files

Table 2 shows which MAC files must be included in the application C-files in order to have access to the entire MAC API.

Table 2. Required MAC Files in Application C-Files

Include file name	Description
DigiType.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.
PhyMacMsg.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.

2.3 MAC API

The MAC API provides a simple and consistent way of interfacing to the Freescale IEEE 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 shows the available API functions for initializing and running the MAC.

Table 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.
status = Mlme_Main()	As the MAC has been designed to be independent of an operating system, part of the MAC must be executed through regular calls to a MAC “main task” implemented as a simple function. The function basically processes data/command messages that are pending in any of the MAC input queues. The function returns `TRUE` if it has more messages to process (that is, it should be called again immediately) and `FALSE` otherwise. See Section 2.4 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 4.

Table 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, NWK_MCPS or APP_ASP. 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 5.

Table 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, MCPS, and ASP 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 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.
aspMessage_t	The data structure of the messages passed from the application to the ASP 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 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 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()`.

2.4 MAC Main Task

Because the MAC is designed to be independent of an operating system, part of the MAC must be executed through regular calls to a MAC "main task" implemented as a simple function. The MAC main task `Mlme_Main()` 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 sequrity 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 function 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 continously 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 behaviour. For optimal beacon operation it is recommended that the `Mlme_Main()` function is polled *at least twice during every superframe*.

2.5 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 IEEE™ 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 IEEE 802.15.4 compliant unit. The MCPS interface carries data related messages (data requests, data indications, data confirmations) and the number of available messages are 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 2.3). 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.

2.5.1 Resetting

Before the Freescale IEEE 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 2.3 and the *802.15.4 MAC/PHY Software Reference Manual*, 802154MPSRM/D, 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 IEEE 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.

2.5.2 Accessing PIB Attributes

The MAC PIB holds all the MAC attributes/variables that are accessible for the application. According to IEEE 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`.
See the *802.15.4 MAC/PHY Software Reference Manual*, 802154MPSRM/D, for a description of how to access all available PIB attributes including Freescale specific PIB attributes.

2.5.3 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 (for a further explanation of energy detection scan see Section 3.2.1) 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.

```
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. 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)
{
  MSG_Queue(&mMlmeNwkInputQueue, pMsg);
  return gSuccess_c;
}
```

As shown in the following code example, dequeuing the messages that were received from either the MCPS, MLME, or ASP interface is done using `MSG_DeQueue()`. Before dequeuing, it makes sense to have called `Mlme_Main()` and then check if there are any messages from the MLME that need to be processed.

```
void main(void)
{
  nwkMessage_t *pMsg;
  uint8_t *pEdList;

  while(1)
  {
    /* Execute the MAC main task function. */
    Mlme_Main();

    /* Try to get a message from the MLME. */
    if(MSG_Pending(&mMlmeNwkInputQueue))
    {
      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. See the *802.15.4 MAC/PHY Software Reference Manual*, 802154MPSRM/D for more details.

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.

2.5.4 MCPS Primitives

Alongside the MLME interface, the MCPS interface processes data related messages (see the figure in Section 2.1). 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(nwkToMcpsMessage_t)` to allocate a MCPS message as shown in the following code example.

```
void App_TransmitUartData_Example(void)
{
  nwkToMcpsMessage_t* pPacket = MSG_AllocType(nwkToMcpsMessage_t);

  if(pPacket != NULL)
  {
    /* If we have a buffer, then get data from the UART. */
    uint8_t msduLength = Uart_Poll(pPacket->msgData.dataReq.msdu);
    if(msduLength)
    {
      /* Data was available in the UART receive buffer. Now create an
         MCPS-Data Request message containing the UART data. */
      pPacket->msgType = gMcpsDataReq_c;
      /* 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(pPacket->msgData.dataReq.dstAddr, coordInfo.coordAddress, 8);
      memcpy(pPacket->msgData.dataReq.srcAddr, myAddress, 8);
      memcpy(pPacket->msgData.dataReq.dstPanId, coordInfo.coordPanId, 2);
      memcpy(pPacket->msgData.dataReq.srcPanId, coordInfo.coordPanId, 2);
      pPacket->msgData.dataReq.dstAddrMode = coordInfo.coordAddrMode;
      pPacket->msgData.dataReq.srcAddrMode = myAddrMode;
      pPacket->msgData.dataReq.msduLength = msduLength;
      /* Request MAC level acknowledgement of the data packet */
      pPacket->msgData.dataReq.txOptions = gTxOptsAck_c;
      /* Give the data packet a handle. The handle is
         returned in the MCPS-Data Confirm message. */
      pPacket->msgData.dataReq.msduHandle = msduHandle++;

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

The application is allowed to allocate multiple data messages using `MSG_AllocType(mcpsMessage_t, ...)` 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 IEEE 802.15.4 Standard, double buffering can also be used for polling. See Section 3.5.1 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);
  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 `main()` function) and the application is responsible of deallocating the MCPS messages.

2.6 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. As with the MLME interface, the ASP interface contains both asynchronous and synchronous messages. In the ASP interface, a request never results in a confirmation message received on the applications ASP SAP handler. It is completed synchronously and the result of the command is checked using the return value of `MSG_Send()`. ASP messages must be allocated using `Msg_AllocType(appToAspMessage_t)` as shown in the following example code.

```
void App_EnterHibernation_Example(void)
{
  appToAspMessage_t Packet;

  /* Create an ASP-Hibernate Request message. */
  Packet.msgType = gAppAspHibernateReq_c;

  /* No further parameters to fill out */

  /* Send the Hibernate Request to the ASP */
  MSG_Send(NWK_ASP, &Packet);
}
```

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-DOZE.request 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 MC13192 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);
  return gSuccess_c;
}
```

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

Chapter 3
Creating an Application

This chapter guides users through the steps required to create a basic, non-beacon network with one coordinator and one device. Throughout this guide there are references to the source code example applications contained in the My_Wireless_App_demo application example. This example can be downloaded from the Freescale Zigbee web-site at www.freescale.com/zigbee. Each source file is a step towards a full wireless UART application. Clear references will be made to the appropriate source file. The guide focuses on the IEEE 802.15.4 Standard and the Freescale specific issues and does not in detail explain the state machine that the code contains. The following source files are available with each new file adding additional functionality until a fully functional application has been developed for both a coordinator and a device. Files with names ending in 'a' are demonstrating coordinator behaviour and files with names ending in 'b' demonstrate device behavior.

- **MyApp_Ex01.c**: Proper initialization of the MAC is demonstrated and the MLME main function is called in the main loop. This application does not yet have any real functionality.
- **MyApp_Ex02.c**: The Energy Detection Scan is used for scanning a range of channels in order to select a specific channel to operate a PAN on.
- **MyApp_Ex03a.c**: This demo application builds on MyApp_Ex02.c. The primary purpose of this application is to demonstrate how to use the Start feature of the MAC in order to configure a PAN coordinator.
- **MyApp_Ex03b.c**: This demo application builds upon MyApp_Ex01.c. The primary purpose of this application is to demonstrate how to use the Active Scan feature of the MAC in order to locate a coordinator that we will associate to later.
- **MyApp_Ex04a.c**: This demo application builds on MyApp_Ex03a.c. In the example we will enable the coordinator to respond to associate requests from devices.
- **MyApp_Ex04b.c**: This demo application builds upon MyApp_Ex03b.c. Now that we have the capability of finding a coordinator, we will associate to one in this demo application.
- **MyApp_Ex05a.c**: This demo application builds on MyApp_Ex04a.c. Now that we have the 802.15.4 features available for building a network, its time to send data.
- **MyApp_Ex05b.c**: This demo application builds upon MyApp_Ex04b.c. In this demo application we will send data to the coordinator that we have previously associated to.

It is recommended that users download the application example and open the My_Wireless_App_demo.mcp file. This file is the guide that shows key points in the source code. However, not all code is explained and the code shown here may be stripped of code that may not be relevant for the given context. It is assumed that users are familiar with Metrowerks

CodeWarrior and know how to build the project for either the Freescale 13191/92 Developer's Starter Kit MC13192DSK or the Freescale evaluation kit MC13193EVK.

3.1 Basic Setup

This section describes the basic setup procedure for any type of device. Refer to the demonstration application in the source file called `MyApp_Ex01.c` for full source code.

3.1.1 Initialization

Before the Freescale IEEE 802.15.4 MAC layer can be accessed, it must be initialized by calling the `Init_802_15_4()` function which will initialize both the MAC and PHY layers as shown in the following example code.

```
Init_802_15_4();
```

The function call initializes internal variables of the modules, resets state machines among other things. Once this function is called, the MAC layer services are available and the MAC and PHY layers are in a known and ready state for further access.

3.1.2 Resetting

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

```
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 (as specified in the IEEE 802,15,4 Standard) or if they are to stay unchanged after the reset. Notice that when requesting the reset primitive, the reset call is synchronous. That is, there is no confirmation message on the reset request and the function `MSG_Send()` simply returns `gSuccess_c` if the reset was successful (it always will be). Because the call is synchronous, 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. Notice that it is not necessary to reset the MAC right after having called `Init_802_15_4()`. `MyApp_Ex01.c` does not contain any example of code for resetting a device using the MLME-RESET.request. However, it is a well-defined way of bringing back the MAC and PHY layers to a known state.

3.1.3 Implementing SAP's

Before being able to compile the project, it is necessary to create the SAP handlers that process the messages from the MAC layer to the next higher layer. For now, just create empty functions and implement the functionality later as shown in this code.

```
/* The following functions are called by the MAC
   to put messages into the Application's queue.
   They need to be defined even if they are not
   used in order to avoid linker errors. */
uint8_t MLME_NWK_SapHandler(nwkMessage_t *pMsg)
{
  /* This only serves as a temporary way of avoiding a compiler warning.
     Later this will be changed so that we return gSuccess_c instead. */
  return pMsg->msgType;
}

uint8_t MCPS_NWK_SapHandler(mcpsToNwkMessage_t *pMsg)
{
  /* This only serves as a temporary way of avoiding a compiler warning.
     Later this will be changed so that we return gSuccess_c instead. */
  return pMsg->msgType;
}

uint8_t ASP_APP_SapHandler(aspToAppMsg_t *pMsg)
{
  /* This only serves as a temporary way of avoiding a compiler warning.
     Later this will be changed so that we return gSuccess_c instead. */
  return pMsg->msgType;
}
```

Typically, the SAP handlers buffer the messages in a queue and then process the messages at regular intervals or whenever appropriate. Section 3.2.1 describes how to process the messages in the SAP handler.

3.2 Starting a New PAN

Once the MAC and PHY layers are initialized (See Section 3.1.1) it is now safe to access the MCPS, ASP, and MLME services of the MAC layer. Start by accessing the MLME. The example code for this is found in the `MyApp_Ex02.c` source file.
The units are now ready to start up a PAN. First, set up a PAN coordinator because all IEEE 802.15.4 Standard PANs must have a PAN coordinator.

3.2.1 Energy Detection Scan

The first task that a PAN coordinator must perform is to choose which radio frequency to use for its PAN. This is called the logical channel. This can be a hard-coded value, but a better method is to choose a channel that is not used by other units. For that purpose, there are primitives that the PAN coordinator can use for scanning all (or selected) channels. This is called the energy detection scan (ED scan). The following code shows this task.

```
uint8_t App_StartScan(uint8_t scanType)
{
  mlmeMessage_t *pMsg;
  mlmeScanReq_t *pScanReq;

  Uart_Print("Sending the MLME-Scan Request message to the MAC...");

  /* Allocate a message for the MLME (We should check for NULL). */
  pMsg = MSG_AllocType(mlmeMessage_t);
  if(pMsg != NULL)
  {
    /* This is a MLME-START.req command */
    pMsg->msgType = gMlmeScanReq_c;
    /* Create the Start request message data. */
    pScanReq = &pMsg->msgData.scanReq;
    /* gScanModeED_c, gScanModeActive_c, gScanModePassive_c, or
gScanModeOrphan_c */
    pScanReq->scanType = scanType;
    /* ChannelsToScan & 0xFF - LSB, always 0x00 */
    pScanReq->scanChannels[0] = (uint8_t)((SCAN_CHANNELS)     & 0xFF);
    /* ChannelsToScan>>8 & 0xFF  */
    pScanReq->scanChannels[1] = (uint8_t)((SCAN_CHANNELS>>8)  & 0xFF);
    /* ChannelsToScan>>16 & 0xFF  */
    pScanReq->scanChannels[2] = (uint8_t)((SCAN_CHANNELS>>16) & 0xFF);
    /* ChannelsToScan>>24 & 0xFF - MSB */
    pScanReq->scanChannels[3] = (uint8_t)((SCAN_CHANNELS>>24) & 0xFF);
    /* Duration per channel 0-14 (dc). T[sec] = (16*960*((2^dc)+1))/1000000.
       A scan duration of 5 on 16 channels approximately takes 8 secs. */
    pScanReq->scanDuration = 5;

    /* Send the Scan request to the MLME. */
    if(MSG_Send(NWK_MLME, pMsg) == gSuccess_c)
    {
      Uart_Print("Done\n");
      return errorNoError;
    }
    else
    {
      Uart_Print("Invalid parameter!\n");
      return errorInvalidParameter;
    }
  }
  else
  {
    /* Allocation of a message buffer failed. */
    Uart_Print("Message allocation failed!\n");
    return errorAllocFailed;
  }
}
```

In this case, the `scanType` parameter for `App_StartScan()` must be set to `gScanModeED_c`. Other types of scanning are explained later in this chapter. If there is any activity on a channel at the time the PAN coordinator scans the channel, this shows up in the result of the scan. A channel that showed no sign of activity at the time of the scan, shows an energy level of approximately 0x00. The higher the number, the more activity that was detected on the channel or the closer the other PAN coordinator is to the scanning PAN coordinator. The previous code example scans all the 16 available channels in the 2.4 GHz band. The parameter `scanChannels` is a bit mask where each bit that is set indicates that this channel must be scanned. Because the 2.4 GHz band contains channel numbers 11 to 26, bits 11 to 26 are set in the `scanChannels` bit mask. Lower channel numbers are for other frequency bands and are ignored if set in the bit bask. Also, the `scanDuration` parameter tells the MAC how long to scan on each channel. The numbers 0 to 14 are valid entries for this parameter. The higher the number, the longer the time spent scanning. The exact scan duration for each channel can be calculated using this equation:

$$\text{Scan duration} = 15.36\ ms * (2^{scanDuration} + 1)$$

The `scanDuration` parameter in the example requests scanning for approximately 0.5 seconds on each of the 16 channels. Once the scan request message has successfully been sent to the MLME, the scanning commences and a scan confirmation (a `nwkMessage_t` struct with `msgType == gNwkScanCnf_c`) is received asynchronously in the SAP handler for the messages from the MLME to the NWK. The following code processes the scan confirmation message.

```
void App_HandleScanEdConfirm(nwkMessage_t *pMsg)
{
  uint8_t n, minEnergy;
  uint8_t *pEdList;

  Uart_Print("Recevied the MLME-Scan Confirm message from the MAC\n");

  /* Get a pointer to the energy detect results */
  pEdList = pMsg->msgData.scanCnf.resList.pEnergyDetectList;

  /* Set the minimum energy to a large value */
  minEnergy = 0xFF;

  /* Select default channel */
  logicalChannel = 11;

  /* Search for the channel with least energy */
  for(n=0; n<16; n++)
  {
    if(pEdList[n] < minEnergy)
    {
      minEnergy = pEdList[n];
      /* Channel numbering is 11 to 26 both inclusive */
      logicalChannel = n + 11;
    }
  }

  /* Print out the result of the ED scan */
  Uart_Print("ED scan returned the following results:\n  [");
  Uart_PrintHex(pEdList, 16, gPrtHexBigEndian_c | gPrtHexCommas_c);
  Uart_Print("]\n\n");
```

```
  /* Print out the selected logical channel */
  Uart_Print("Based on the ED scan the logical channel 0x");
  Uart_PrintHex(&logicalChannel, 1, 0);
  Uart_Print(" was selected\n");

  /* The list of detected energies must be freed. */
  MSG_Free(pEdList);
}
```

Assuming that the scanning was successful, (`status == gSuccess_c`) and that the `nwkMessage_t` struct is pointed to by a pointer `pMsg` the `pEdList = pMsg->msgData.scanCnf.resList.pEnergyDetectList` points to a byte array of 16 bytes. One byte for each channel. `pEdList[0]` contains the result for channel 11, `pEdList[1]` contains the result for channel 12 and so on. There will only be a valid result for the channels that were requested to scan for, so a 0x00 result means that either the channel is probably available or the channel was not scanned. It is important to remember the channels that the scan was requested for.
Once the results of the energy detection scan are received, look at the results from all the channels. Remember, a request was made to scan all the channels and to pick the channel with the lowest energy level. Store this channel number in a global variable called `logicalChannel`.
Do not forget to free not only the scan confirm message (this is done in `main()` in `MyApp_Ex02.c`) but also the data structures pointed to by `pEdList` – and free `pEdList` first (unless you have a local copy of `pEdList` that you can use for freeing the list of energy levels) .

NOTE

Just because the ED scan returned a result that showed no activity on a channel it does not mean that another PAN coordinator is not using this channel. It only means that no transaction(s) took place between this PAN coordinator and any of its devices while performing the ED scan. Scanning for a longer time increases the possibility that another PAN coordinator is on the channel and is detected, but it is not guaranteed!

Notice that in `MyApp_Ex02.c` there has been added code, as shown in the following code example, for queuing incoming MLME messages and decoupling the MLME from the application.

```
/* Application input queues */
anchor_t mMlmeNwkInputQueue;

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

Also, in the main loop of the application there is added code for processing incoming messages from that queue as shown in this code example.

```
/* Try to get a message from MLME */
if(MSG_Pending(&mMlmeNwkInputQueue))
  pMsgIn = MSG_DeQueue(&mMlmeNwkInputQueue);
else
  pMsgIn = NULL;

… /* Process message if pMsgIn!=NULL */

if(pMsgIn)
{
  /* Messages from the MLME must always be freed. */
  MSG_Free(pMsgIn);
}
```

By implementing the MLME to NWK SAP handler this way, the MLME and NWK/APP are completely decoupled. That is, the SAP handler only queues the message but does not do any processing of the message. In this way, the MLME returns from the call as fast as possible and the call stack of the MCU is excercised as little as possible reducing the risk of getting a call stack overflow.

3.2.2 Choosing Short Address

Now the coordinator must assign itself a short address. All units come with a pre-assigned long address, but a short address must be assigned before starting the PAN. Otherwise, the start request will fail. Because the PAN coordinator is the first unit to participate in its own PAN, it can chose any short address for itself. Once the short address is chosen, it is usually hard-coded, it must be set by setting the macShortAddress PIB attribute. This is done in `App_StartCoordinator()` in `MyApp_Ex03a.c` as shown in the following example code.

```
uint8_t App_StartCoordinator(void)
{
  /* Message for the MLME will be allocated and attached to this pointer */
  mlmeMessage_t *pMsg;

  Uart_Print("Sending the MLME-Start Request message to the MAC...");

  /* Allocate a message for the MLME (We should check for NULL). */
  pMsg = MSG_AllocType(mlmeMessage_t);
  if(pMsg != NULL)
  {
    /* Pointer which is used for easy access inside the allocated message */
    mlmeStartReq_t *pStartReq;
    /* Return value from MSG_send - used for avoiding compiler warnings */
    uint8_t ret;
    /* Boolean value that will be written to the MAC PIB */
    uint8_t boolFlag;

    /* Set-up MAC PIB attributes. Please note that Set, Get,
       and Reset messages are not freed by the MLME. */

    /* We must always set the short address to something
       else than 0xFFFF before starting a PAN. */
    pMsg->msgType = gMlmeSetReq_c;
    pMsg->msgData.setReq.pibAttribute = gMacPibShortAddress_c;
```

```
    pMsg->msgData.setReq.pibAttributeValue = (uint8_t *)shortAddress;
    ret = MSG_Send(NWK_MLME, pMsg);

    … /Lot of other stuff going on here */

  }
  else
  {
    /* Allocation of a message buffer failed. */
    Uart_Print("Message allocation failed!\n");
    return errorAllocFailed;
  }
}
```

In the previous code example, the short address of the PAN coordinator is set to the value of `shortAddress` which is elsewhere set to `0xCAFE`. As with the reset request, the MLME-SET.request is also completed synchronously. So, if `ret == gSuccess_c` after calling `MSG_Send()`, the PIB attribute was set successfully. The `pibAttributeValue` parameter is not freed by the MLME.
The short address must be different from 0xFFFF. A short address of 0xFFFE tells the coordinator to use its long address in all transactions. If the short address is set to anything different from 0xFFFF and 0xFFFE, the PAN coordinator uses this address instead of its long address and thus sends shorter packets. A long address is 8 bytes long, a short address is 2 bytes long.

3.2.3 Choosing PAN ID

After selecting the logical channel, the last thing required before a PAN coordinator can start a PAN, is for the coordinator to select an identification number for the PAN which is called the PANid. Trusting that there really is no other PAN using the same logical channel, the new PAN coordinator can freely choose a PANid. However, users may want to perform an active scan (see Section 3.3.1) of the channel to see if there really are no other PAN coordinators using the same logical channel. If so, the PANid used must be different from the one used by the other PAN coordinator on the same logical channel. In the source example file `MyApp_Ex03a.c` it is assumed that no other IEEE 802.15.4 PAN is present and it sets the global variable `panId = 0xBEEF`.

3.2.4 Starting a PAN

After choosing a logical channel, PANid, and short address, it is time to start up the PAN using the MLME_START.request primitive.

The `panCoordinator` parameter indicates whether the start request is to start up a PAN for a PAN coordinator or for a coordinator without PAN coordinator capability. For an explanation of the difference, see the IEEE 802.15.4 Standard. Because this example is starting up a PAN coordinator, this parameter is set to 0x01, where 0x01 means yes, and 0x00 means no.

The `beaconOrder` and `superFrameOrder` parameters are set to 0x0F because users want to start a non-beacon network. See the 802.15.4 Standard about how to start a beacon network. Any combination of `beaconOrder` and `superFrameOrder` where `beaconOrder` is set to 0x0F creates a non-beacon network.

The `batteryLifeExt` parameter is ignored for now and is set to 0x00.

The `coordRealignment` parameter tells the MLME whether it should send out a coordinator realignment command prior to starting up the PAN. For now, users should set this to 0x00 (no coordinator realignment command) because it is not relevant for this example.

Finally, the `securityEnable` parameter tells the MLME if it should apply any security to the transactions taking place over the air. For now, users should leave this set to 0x00 (no security).

`App_StartCoordinator()` in `MyApp_Ex03a.c` contains the code for starting a PAN.

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

Uart_Print("Sending the MLME-Start Request message to the MAC...");

/* Allocate a message for the MLME (We should check for NULL). */
pMsg = MSG_AllocType(mlmeMessage_t);
if(pMsg != NULL)
{
/* Pointer which is used for easy access inside the allocated message */
mlmeStartReq_t *pStartReq;
/* Return value from MSG_send - used for avoiding compiler warnings */
uint8_t ret;
/* Boolean value that will be written to the MAC PIB */
uint8_t boolFlag;

… /* Various MAC PIB attributes are set up. */

/* This is a MLME-START.req command */
pMsg->msgType = gMlmeStartReq_c;

/* Create the Start request message data. */
pStartReq = &pMsg->msgData.startReq;
/* PAN ID - LSB, MSB. The example shows a PAN ID of 0xBEEF. */
memcpy(pStartReq->panId, (void *)panId, 2);
/* Logical Channel - the default of 11 will be overridden */
pStartReq->logicalChannel = logicalChannel;
/* Beacon Order - 0xF = turn off beacons */
pStartReq->beaconOrder = 0x0F;
/* Superframe Order - 0xF = turn off beacons */
pStartReq->superFrameOrder = 0x0F;
/* Be a PAN coordinator */
pStartReq->panCoordinator = TRUE;
/* Dont use battery life extension */
pStartReq->batteryLifeExt = FALSE;
```

```
/* This is not a Realignment command */
pStartReq->coordRealignment = FALSE;
/* Dont use security */
pStartReq->securityEnable = FALSE;

/* Send the Start request to the MLME. */
if(MSG_Send(NWK_MLME, pMsg) == gSuccess_c)
{
  Uart_Print("Done\n");
  return errorNoError;
}
else
{
  /* One or more parameters in the Start Request message were invalid. */
  Uart_Print("Invalid parameter!\n");
  return errorInvalidParameter;
}
```

A start confirmation is received asynchronously on the SAP that handles the messages from the MLME to the NWK. If the PAN was started successfully, a status code of `gSuccess_c` is returned. The MLME-START.confirm message is processed in `MyApp_Ex03a.c` in the state `stateStartCoordinatorWaitConfirm`. As shown in the following code.

```
case stateStartCoordinatorWaitConfirm:
  /* Stay in this state until the Start confirm message
     arrives, and then goto the Listen state. */
  ret = App_WaitMsg(pMsgIn, gNwkStartCnf_c);
  if(ret == errorNoError)
  {
    Uart_Print("Started the coordinator with PAN ID 0x");
    Uart_PrintHex((uint8_t *)panId, 2, 0);
    Uart_Print(", and short address 0x");
    Uart_PrintHex((uint8_t *)shortAddress, 2, 0);
    Uart_Print(".\n");
    state = stateListen;
  }
  break;
```

After successfully starting a PAN, the macRxOnWhenIdle PIB has been set to 0x01. This means that whenever the PAN coordinator is not transmitting a packet, it is listening for incoming packets. This behavior is Freescale proprietary and is implemented this way because it is not logical to start a PAN and then not start listening for incoming association requests or beacon requests from devices performing active scans. However, if this behavior is unwanted, the macRxOnWhenIdle PIB can be set back to its default value of 0x00.
All that needs to be done on the PAN coordinator is that the PIB attribute macAssociationPermit must be set to 0x01 in order to ensure that devices are allowed to associate to the PAN coordinator. If this PIB is left untouched, or at any time set to 0x00, any incoming association requests from a device will be ignored by the MAC (see `App_StartCoordinator()`) as shown in the following code.

```
/* We must set the Association Permit flag to TRUE
   in order to allow devices to associate to us. */
pMsg->msgType = gMlmeSetReq_c;
pMsg->msgData.setReq.pibAttribute = gMacPibAssociationPermit_c;
boolFlag = TRUE;
pMsg->msgData.setReq.pibAttributeValue = &boolFlag;
ret = MSG_Send(NWK_MLME, pMsg);
```

As was the case when setting the macShortAddress PIB, the `ret` variable contains `gSuccess_c` if the PIB was successfully set.
In the source code example in `MyApp_Ex03a.c`, the allocated message `pMsg` was used repeatedly for setting PIB attributes and for starting the PAN. When setting PIB attributes the message is not freed by the MLME so it can be used again for things like setting another PIB attribute or starting the PAN. But as soon as `pMsg` is used for requesting the startup of a PAN, the message is no longer valid in the application context. Now the message is owned by the MLME and is freed by the MLME. So, in `App_StartCoordinator()`, `pMsg` is never freed because this is eventually done by the MLME.

3.3 Joining a PAN

The next step in creating a PAN is to associate a device with the PAN coordinator that was just started up. The association between the device and the PAN coordinator is necessary because the PAN coordinator is the only device type that is allowed to assign short addresses to devices and coordinators and a device must have a short address assigned. Also, during the (successful) association procedure, the logical channel and PANid that the device is going to use when transmitting data is set. The following code examples are taken from `MyApp_Ex03b.c`.

3.3.1 Active Scan

Before associating to a PAN coordinator, the device first needs to find a PAN coordinator that is accepting incoming association requests. For that purpose, the device must use the active scan feature. This feature is initiated much the same way as the energy detection (ED) scan and users can reuse the function `App_StartScan()` as described in Section 3.2.1. The only difference is that the `scanType` parameter must now be set to `gScanModeActive_c`.
The device could also do an ED scan prior to doing the active scan in order to estimate which logical channels would be worth doing active scan on. However, because the PAN that was set up by the PAN coordinator is non-beacon and thus no air-traffic is available to detect in the ED scan because there is also no other data on the air, users should not do an ED scan on the device.
After receiving the active scan request, the MLME starts sending out beacon requests on the requested channels (in the `scanChannels` parameter) and starts listening for beacons from (PAN) coordinators for as long as indicated in the `scanDuration` parameter. Basically, sending a beacon request is like asking, *“Are there any PAN coordinators out there that have a PAN on this channel?”* and a PAN coordinator sending back a beacon means, “*Yes, I have a PAN, here is my PAN information”*.
Because the device does not know which channel the PAN coordinator chose after performing the ED scan, the device will be scanning all channels. Assuming that only the PAN coordinator that was just started is in the vicinity of our device and it did see the beacon request, a scan confirm with the status == gSuccess_c and resultListSize == 1 is returned. If no answer was received, then status == gNoBeacon_c.
After receiving the scan confirmation, now look at the `pPanDescriptor` pointer in the scan confirmation message and the data it is pointing to. The `pPanDescriptor` pointer points to an

array of structures of type panDescriptor_t as many as are indicated in the resultListSize parameter as shown in the following code example.

```
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;
```

Using the list of panDescriptor_t structure pointed to by pPanDescriptor, the device can decide which coordinator to associate to. For now, skip the parameters that are irrelevant for this example because this example does not run a beacon PAN and does not use security. The securityUse, aclEntry , securityFailure, gtsPermit, and timeStamp parameters are not used in this example.
The coordAddrMode denotes whether the coordinator is using its long address or is using a short address. If set to gAddrModeShort_c the PAN, the coordinator uses a 16 bit short address and only the first two bytes in coordAddress are valid and contains (in little endian mode) the short address of the PAN coordinator. If coordAddrMode is set to gAddrModeLong_c all 8 bytes of the coordAddress parameter are valid and contains (in little endian mode) the full address of the PAN coordinator. In this example, coordAddrMode equals gAddrModeShort_c and coordAddress equals 0xCAFE.
The logicalChannel denotes the logical channel where the PAN coordinator can be found. This value can vary in this example, but it is probably set to 0x0B. See Section 3.2.1 for more details. The coordPanId contains the PANid of the PAN that the PAN coordinator has set up. In this example, it is 0xBEEF. See Section 3.2.4. The linkQuality contains a value in the range of 0x00 to 0xFF – the larger the value, the better the signal strength from the PAN coordinator. In most cases this means the closer the PAN coordinator is to the device.
The last parameter that the device must look at is the superFrameSpec parameter. This parameter contains the information as shown in Table 9.

Table 9. superFrameSpec Parameter Contents

Bits: 0-3	4-7	8-11	12	13	14	15
Beacon order	Superframe order	Final CAP slot	Battery life extension	Reserved	PAN coordinator	Association permit

The bit definition is covered later in this Chapter. However, Beacon order bit and Superframe order bit must correspond to the beacon order and superframe order used when starting the PAN coordinator. See Section 3.2.4. In this non-beacon example, they are both 0xF. The PAN coordinator bit must also be set as this bit is set as shown in Section 3.2.4. The last bit that is required to look at is the Association permit bit. This bit indicates if the PAN coordinator will process any incoming association requests. If set, it will process the requests, if not set, the association requests are ignored. This bit follows the macAssociatePermit PIB attribute of the coordinator as shown in Section 3.2.4.
The following code example from `MyApp_Ex03b.c` shows how an incoming scan confirmation on an active scan is handled.

```
uint8_t App_HandleScanActiveConfirm(nwkMessage_t *pMsg)
{
  uint8_t panDescListSize   = pMsg->msgData.scanCnf.resultListSize;
  panDescriptor_t *pPanDesc = pMsg-
>msgData.scanCnf.resList.pPanDescriptorList;
  uint8_t rc = errorNoScanResults;

  /* Check if the scan resulted in any coordinator responses. */
  if(panDescListSize != 0)
  {
    /* Initialize link quality to very poor. */
    uint8_t i, bestLinkQuality = 0;

    /* Check all PAN descriptors. */
    for(i=0; i<panDescListSize; i++, pPanDesc++)
    {
      /* Only attempt to associate if the coordinator
         accepts associations and is non-beacon. */
      if( ( pPanDesc->superFrameSpec[1] & gSuperFrameSpecMsbAssocPermit_c) &&
          ((pPanDesc->superFrameSpec[0] & gSuperFrameSpecLsbBO_c) == 0x0F) )
      {
        /* Find the nearest coordinator using the link quality measure. */
        if(pPanDesc->linkQuality > bestLinkQuality)
        {
          /* Save the information of the coordinator candidate. If we
             find a better candiate, the information will be replaced. */
          memcpy(&coordInfo, pPanDesc, sizeof(panDescriptor_t));
          bestLinkQuality = pPanDesc->linkQuality;
          rc = errorNoError;
        }
      }
    }
  }
  /* ALWAYS free the PAN descriptor list */
  MSG_Free(pMsg->msgData.scanCnf.resList.pPanDescriptorList);

  return rc;
}
```

As this code example shows, the device stores the PAN descriptor for the chosen coordinator in a global variable called `coordInfo`. The contents of this PAN descriptor will be used in the next section when the device requests to be associated with the coordinator.
Remember to free not only the scan confirm message (this is done in `main()` in the example application) but also the data structures pointed to by `pMsg->msgData.scanCnf.resList.pPanDescriptorList` – and free `pMsg-`

>msgData.scanCnf.resList.pPanDescriptorList first. That is, unless users have a local copy that they can use for freeing the list of PAN descriptors.

3.3.2 Associating to a PAN

From the PAN descriptor that was returned by the PAN coordinator, users can see that the PAN coordinator does accept incoming association requests and there is also a short address and a PANid of its PAN. Next, users can associate the device to the PAN coordinator. Use the association request message (see MyApp_Ex04b.c) to accomplish this as shown in the following code example.

```
uint8_t App_SendAssociateRequest(void)
{
  mlmeMessage_t *pMsg;
  mlmeAssociateReq_t *pAssocReq;

  Uart_Print("Sending the MLME-Associate Request message to the MAC...");

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

    /* Create the Associate request message data. */
    pAssocReq = &pMsg->msgData.associateReq;

    /* Use the coordinator info we got from the Active Scan. */
    memcpy(pAssocReq->coordAddress, coordInfo.coordAddress, 8);
    memcpy(pAssocReq->coordPanId,   coordInfo.coordPanId, 2);
    pAssocReq->coordAddrMode      = coordInfo.coordAddrMode;
    pAssocReq->logicalChannel     = coordInfo.logicalChannel;
    pAssocReq->securityEnable     = FALSE;
    /* We want the coordinator to assign a short address to us. */
    pAssocReq->capabilityInfo     = gCapInfoAllocAddr_c;

    /* Send the Associate Request to the MLME. */
    if(MSG_Send(NWK_MLME, pMsg) == gSuccess_c)
    {
      Uart_Print("Done\n");
      return errorNoError;
    }
    else
    {
      /* One or more parameters in the message were invalid. */
      Uart_Print("Invalid parameter!\n");
      return errorInvalidParameter;
    }
  }
```

```
  else
  {
    /* Allocation of a message buffer failed. */
    Uart_Print("Message allocation failed!\n");
    return errorAllocFailed;
  }
}
```

Most of the parameters in the association request message are recognizable from previous sections. One parameter that is new is the `capabilityInfo` parameter. This bit mask is described in Table 10.

Table 10. capabilityInfo Parameter

Bit 0	Bit 1	Bit 2	Bit 3	Bit 4-5	Bit 6	Bit 7
Alternate PAN coordinator	Device type	Power source	Receiver on when idle	Reserved	Security capability	Allocate address

Table 11. capabilityInfo Field Definitions

Field Name	Setting
Alternate PAN coordinator	1 = If device is capable of being a PAN coordinator 0 = If device is not capable of being a PAN coordinator
Device Type	1 = If device is an Radio Frequency Device (RFD) 0 = If device is not an Radio Frequency Device (RFD)
Power Source	1 = If device is receiving power from AC power 0 = If device is not receiving power from AC power
Receiver on when idle	1= If device does not disable its receiver during idle periods 0 = If device does disable its receiver during idle periods
Reserved	
Secure Capability	1 = If device is capable of sending and receiving MAC frames using the security suite. 0 = If device is not capable of sending and receiving MAC frames using the security suite. Field is one bit in length.
Allocate Address	1 = If device wants the coordinator to allocate a short address as a result of the association procedure. 0 = The special short address (0xFFFE) is allocated to the device and returned through the association response command. In this case, the device communicates on the PAN using only its 64 bit extended address. Field is one bit in length.

In this example, the device that users are trying to associate will only have the allocate address subfield set to 1, all others are set to 0. The capabilityInfo parameter is set to `gCapInfoAllocAddr_c` (0x80).
After sending the association request message to the MLME, an association request is sent from the device to the PAN coordinator. Assuming that the PAN coordinator receives, accepts, and

responds positively to the association request by sending an association response as described in Section 3.4, the application on the device receives an association confirmation with status `gSuccess_c` and the `assocShortAddress` parameter set to the short address that the PAN coordinator has assigned to the device. In this case the device short address has been set to 0x0001 (`assocShortAddress[0] = 0x00, assocShortAddress[1] = 0x01`) as shown in the following example code.

```
void App_HandleAssociateConfirm(nwkMessage_t *pMsg)
{
  /* This is our own extended address (MAC address). It cannot be modified. */
  extern const uint8_t aExtendedAddress[8];

  /* If the coordinator assigns a short address of 0xfffe then,
     that means we must use our own extended address in all
     communications with the coordinator. Otherwise, we use
     the short address assigned to us. */
  if( (pMsg->msgData.associateCnf.assocShortAddress[0] >= 0xFE) &&
      (pMsg->msgData.associateCnf.assocShortAddress[1] == 0xFF) )
  {
    myAddrMode = 3;
    memcpy(myAddress, (void *)aExtendedAddress, 8);
  }
  else
  {
    myAddrMode = 2;
    memcpy(myAddress, pMsg->msgData.associateCnf.assocShortAddress, 2);
  }
}
```

As the example code shows, the address assigned to the device by the coordinator is stored in `myAddress` and the addressing mode that is used is stored in `myAddrMode`. If the coordinator assigns the short address 0xFFFE to the device, the device must always use its long (8 byte) address.
In this particular scenario there is no checking the `pMsg->msgData.associateCnf.status` parameter because the coordinator always accepts incoming association requests. However, in a more extensive application it would be necessary to check on this parameter.

3.4 Adding a Device to the PAN

Until now this example has ignored what happens on the PAN coordinator when the association request from the device is received by the PAN coordinator and passed up to the application as an association indication. The association indication contains the extended device address (in the `deviceAddress` parameter) of the associating device and a copy of the capability information that the device passed to the MLME in the association request message. The capability information is stored in the `capabilityInfo` parameter as described in Section 3.3.2.

As the allocate address bit is set in the `capabilityInfo` parameter, the PAN coordinator must assign a short address to the device. Because the PAN coordinator is the only unit in the PAN that is allowed to assign short addresses to a device it can freely choose anything in the 0x0000-0xFFFD range. 0xFFFF is an invalid short address and 0xFFFE is the short address that must be assigned to devices that do not request a short address or that always must use a long address. In this example the incoming association indication is accepted and the short address 0x0001 is chosen for the device. As shown in the following code, this message must be sent to the MLME.

```
uint8_t App_SendAssociateResponse(nwkMessage_t *pMsgIn)
{
  mlmeMessage_t *pMsg;
  mlmeAssociateRes_t *pAssocRes;

  Uart_Print("Sending the MLME-Associate Response message to the MAC...");

  /* Allocate a message for the MLME */
  pMsg = MSG_AllocType(mlmeMessage_t);
  if(pMsg != NULL)
  {
    /* This is a MLME-ASSOCIATE.res command */
    pMsg->msgType = gMlmeAssociateRes_c;

    /* Create the Associate response message data. */
    pAssocRes = &pMsg->msgData.associateRes;

    /* Assign a short address to the device. In this example we simply
       choose 0x0001. Though, all devices and coordinators in a PAN must have
       different short addresses. However, if a device do not want to use
       short addresses at all in the PAN, a short address of 0xFFFE must
       be assigned to it. */
    if(pMsgIn->msgData.associateInd.capabilityInfo & gCapInfoAllocAddr_c)
    {
      /* Assign a unique short address less than 0xfffe if the device requests
so. */
      pAssocRes->assocShortAddress[0] = 0x01;
      pAssocRes->assocShortAddress[1] = 0x00;
    }
    else
    {
      /* A short address of 0xfffe means that the device is granted access to
         the PAN (Associate successful) but that long addressing is used.*/
      pAssocRes->assocShortAddress[0] = 0xFE;
      pAssocRes->assocShortAddress[1] = 0xFF;
    }
    /* Get the 64 bit address of the device requesting association. */
```

```
    memcpy(pAssocRes->deviceAddress, pMsgIn-
>msgData.associateInd.deviceAddress, 8);
    /* Association granted. May also be gPanAtCapacity_c or
gPanAccessDenied_c. */
    pAssocRes->status = gSuccess_c;
    /* Do not use security */
    pAssocRes->securityEnable = FALSE;

    /* Save device info. */
    memcpy(deviceShortAddress, pAssocRes->assocShortAddress, 2);
    memcpy(deviceLongAddress,  pAssocRes->deviceAddress,     8);

    /* Send the Associate Response to the MLME. */
    if(MSG_Send(NWK_MLME, pMsg) == gSuccess_c) …
```

This concludes the association procedure for the PAN coordinator and completes the association procedure. Had users not accepted the association request, the `status` parameter would have been set to `gPanAccessDenied_c`. If the PAN coordinator does not want to see any incoming association requests, it can set the PIB attribute macAssociatePermit to 0x00 (FALSE). See Section 3.2.4 for details. As was the case for the device, the PAN coordinator must store the extended 8 byte address of the device contained in the `deviceAddress` parameter so that it can later disassociate from the device if it needs to.
As an indication of a successful association procedure the coordinator receives a MLME-COMM-STATUS.indication message which must be managed as shown in the following code example.

```
uint8_t App_HandleMlmeInput(nwkMessage_t *pMsg)
{
  if(pMsg == NULL)
    return errorNoMessage;

  /* Handle the incoming message. The type determines the sort of
processing.*/
  switch(pMsg->msgType) {
  case gNwkAssociateInd_c:
    Uart_Print("Received an MLME-Associate Indication from the MAC\n");
    /* A device sent us an Associate Request. We must send back a response.
*/
    return App_SendAssociateResponse(pMsg);
    break;

  case gNwkCommStatusInd_c:
    /* Sent by the MLME after the Association Response has been transmitted.
*/
```

```
Uart_Print("Received an MLME-Comm-Status Indication from the MAC\n");
    break;
  }
  return errorNoError;
}
```

The example code does not expect that the association procedure fails and does not check on the status code of the MLME-COMM-STATUS.indication. However, the MLME-COMM-STATUS.indication also contains other parameters such as the addresses of the coordinator and the device, their addressing modes, and the PANid used by the association procedure. See the *802.15.4 MAC/PHY Software Reference Manual* , 802154MPSRM/D for more information.

3.5 Data Transfer

Now that the PAN coordinator and the device are associated, data must be transferred between the units.

In IEEE 802.15.4 Standard PANs, most communications are driven by the devices in a network. The devices are typically battery powered and must be able to control the data flow in order to optimize battery life. This is accomplished by the device polling for data from the coordinator and transmitting the data directly to the coordinator. The coordinator only sends data to a device when it knows it is listening, for example, when a device has requested data.

3.5.1 Direct Data

In many user scenarios, there will be one PAN coordinator with several devices associated to it and the PAN coordinator will be powered with AC power, where the devices will be powered by a battery. Also, the devices will decide when to transfer data. Therefore, the normal set up is to have the PAN coordinator listen for data all the time (except when it is transmitting) and have the devices initiate all data transfers. This behavior is implemented in the PAN coordinator by setting the macRxOnWhenIdle PIB attribute to 0x01 (TRUE - the default value is 0x00 (FALSE) which corresponds to not listening). This PIB is automatically set to 0x01 when the coordinator starts the PAN. See Section 3.2.4 for details.

Sending data using the MCPS interface is achieved similar to sending commands on the MLME interface. However, there is one major difference. The data may be required to be sent using high bandwidth. On the MLME interface, there can only be one outstanding command packet at a time. Only when a confirmation message has been received is the application allowed to send a new command. However, on the MCPS interface, there can be multiple outstanding data packets (as many as can be allocated by the application) at a time, keeping the throughput at a maximum.

In non-beacon mode, direct data, as opposed to indirect data, is sent immediately and is sent from the device to the coordinator. The device can send any time because the coordinator is always listening. The coordinator is AC powered and does not need to conserve battery power.

As shown in the following code example, `MyApp_Ex05b.c` shows how to send data from the device to the coordinator and achieve maximum throughput while sending data.

```
void App_TransmitUartData(void)
{
  /* Use multi buffering for increased TX performance. It does not really
     have any effect at a UART baud rate of 19200bps but serves as an
     example of how the throughput may be improved in a real-world
     application where the data rate is of concern. */
  if( (numPendingPackets < MAX_PENDING_DATA_PACKETS) && (pPacket == NULL) )
  {
    /* If the maximum number of pending data buffes is below maximum limit
       and we do not have a data buffer already then allocate one. */
    pPacket = MSG_AllocType(nwkToMcpsMessage_t);
  }

  if(pPacket != NULL)
  {
    /* If we have a buffer, then get data from the UART. */
    uint8_t msduLength = Uart_Poll(pPacket->msgData.dataReq.msdu);
    if(msduLength)
    {
      /* Data was available in the UART receive buffer. Now create an
         MCPS-Data Request message containing the UART data. */
      pPacket->msgType = gMcpsDataReq_c;
      /* 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(pPacket->msgData.dataReq.dstAddr, coordInfo.coordAddress, 8);
      memcpy(pPacket->msgData.dataReq.srcAddr, myAddress, 8);
      memcpy(pPacket->msgData.dataReq.dstPanId, coordInfo.coordPanId, 2);
      memcpy(pPacket->msgData.dataReq.srcPanId, coordInfo.coordPanId, 2);
      pPacket->msgData.dataReq.dstAddrMode = coordInfo.coordAddrMode;
      pPacket->msgData.dataReq.srcAddrMode = myAddrMode;
      pPacket->msgData.dataReq.msduLength = msduLength;
      /* Request MAC level acknowledgement of the data packet */
      pPacket->msgData.dataReq.txOptions = gTxOptsAck_c;
      /* Give the data packet a handle. The handle is
         returned in the MCPS-Data Confirm message. */
      pPacket->msgData.dataReq.msduHandle = msduHandle++;

      /* Send the Data Request to the MCPS */
      NR MSG_Send(NWK_MCPS, pPacket);
      /* Prepare for another data buffer */
      pPacket = NULL;
      numPendingPackets++;
    }
  }
}
```

As this code example shows, the function `TransmitUartData()` must be called at regular intervals (typically when also calling `Mlme_Main()`) in order to keep the throughput at a maximum. Also, the number of outstanding data packets in the example has been limited by a constant `MAX_PENDING_DATA_PACKETS`. To make this limitation is a matter of design and may influence the throughput if it is set too low depending on the data transfer scenario.
Most of the parameters in the data request message have already been explained in previous sections and basically just contain addresses and addressing modes. However, the `txOptions`, `msduHandle`, and `msduLength` require some explanation.

- The `msduHandle` parameter is a unique identifier for the data packet. This identifier can be used when receiving the MCPS-DATA.confirm message to identify which data

packet the MCPS-DATA.message refers to. The `msduHandle` parameter must be chosen so that it can uniquely identify the outstanding data packets. The value is only typically increased between each data packet that is sent as shown throughout this example.

- The `msduLength` parameter indicates how many valid data bytes the data buffer `pPacket->msgData.dataReq.msdu` contains. The `pPacket->msgData.dataReq.msdu` contains the data that must be transferred, up to 102 bytes.
- The `txOptions` parameter is a bit mask that tells the MCPS how to transfer the data. In this example, only the data packet was required to be acknowledged along with no GTS, no indirect transmission, and no security options enabled.

The `txOptions` bit mask is encoded as shown in Table 12.

Table 12. txOptions Parameter Encoding

Bit 0	Bit 1	Bit 2	Bit 3	Bits 4-7
Acknowledge required	GTS transmission	Indirect transmission	Security enabled	0

The data sent by the device to the coordinator is sent as soon as possible because direct transmission is used. The coordinator is always listening and must acknowledge the reception of the data packet if it was received successfully. If the coordinator was not listening, the device will get a MCPS-DATA.confirm message indicating that no acknowledge was received. However, in the demo application it is assumed that the data was transferred successfully and a counter that keeps track of the number of outstanding data packets is decreased.

```
void App_HandleMcpsInput(mcpsToNwkMessage_t *pMsgIn)
{
  switch(pMsgIn->msgType)
  {
    /* The MCPS-Data confirm is sent by the MAC to the network
       or application layer when data has been sent. */
  case gMcpsDataCnf_c:
    if(numPendingPackets)
      numPendingPackets--;
    break;
  }
}
```

On the coordinator side (see `MyApp_Ex05a.c`) the data that the device sent is received much the same way, only it is receiving MCPS-DATA. indications instead of MCPS-DATA.confirms. The coordinator just passes the received data on to the UART.

```
void App_HandleMcpsInput(mcpsToNwkMessage_t *pMsgIn)
{
  switch(pMsgIn->msgType)
  {
  case gMcpsDataInd_c:
    /* The MCPS-Data indication is sent by the MAC to the network
       or application layer when data has been received. We simply
       copy the received data to the UART. */
    Uart_Tx(pMsgIn->msgData.dataInd.msdu, pMsgIn->msgData.dataInd.msduLength);
```

```
    break;
  }
}
```

For data transfer to actually take place, the SAP handler for the MCPS interface on both the coordinator and the device must now be implemented just like the MLME SAP handler did.

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

The MCPS SAP handler uses its own queue and does not share queue with the MLME SAP handler because the message formats are different and cannot easily be distinguished by looking at the message. The processing of the messages in the MCPS queue starts in the main loop of the application and on both the coordinator and device it follows the pattern as shown in the following example code.

```
if(MSG_Pending(&mMcpsNwkInputQueue))
{
  pMsgIn = MSG_DeQueue(&mMcpsNwkInputQueue);
  … /* Process the MCPS packet appripriately */
  /* ALWAYS free messages from MCPS */
  MSG_Free(pMsgIn);
}
```

NOTE

The MCPS message *must* be freed by the application.

3.5.2 Indirect Data

Sending data from the coordinator to the device is different than sending data from the device to the coordinator because the device is not necessarily listening. The device is usually battery powered and may shut down its transceiver at any time. This requires that the coordinator send its data indirectly. That is, the coordinator sends its data and the data is buffered until the device polls for it. The data is not sent immediately. The data message is created and set as was shown in Section 3.5.1 except from the `txOptions` parameter (see `MyApp_Ex06a.c`) as shown in the following example code.

```
/* Request MAC level acknowledgement, and
   indirect transmission of the data packet */
pPacket->msgData.dataReq.txOptions = gTxOptsAck_c | gTxOptsIndirect_c;
```

The coordinator needs to set the `gTxOptsIndirect_c` bit in the `txOptions` parameter. If this bit is not set, the data is transmitted directly and immediately. This means that the data would probably be lost as the device is probably not listening. In non-beacon mode, the coordinator typically uses indirect data transmission.
By using the `gTxOptsAck_c` bit, the coordinator (and the device) will not only get a MCPS-DATA.confirm message stating that the packet was sent successfully, but they will also get a

message if the data packet was not acknowledged and hence not received by the intended receiver. This allows for an opportunity to retransmit the data packet.
Because the coordinator sends its data indirectly, the device must poll the data out of the coordinator when the device decides that it is in a state where is it ready to receive and process a message from the coordinator. This is done using the MLME-POLL.request message (see MyApp_Ex06b.c) as shown in the following example code.

```
/* This is an MLME-POLL.req command. */
mlmeMessage_t *pMlmeMsg = MSG_AllocType(mlmeMessage_t);
if(pMlmeMsg)
{
  /* Create the Poll Request message data. */
  pMlmeMsg->msgType = gMlmePollReq_c;

  /* Use the coordinator information we got from the Active Scan. */
  memcpy(pMlmeMsg->msgData.pollReq.coordAddress, coordInfo.coordAddress, 8);
  memcpy(pMlmeMsg->msgData.pollReq.coordPanId, coordInfo.coordPanId, 2);
  pMlmeMsg->msgData.pollReq.coordAddrMode = coordInfo.coordAddrMode;
  pMlmeMsg->msgData.pollReq.securityEnable = FALSE;

  /* Send the Poll Request to the MLME. */
  if(MSG_Send(NWK_MLME, pMlmeMsg) == gSuccess_c)
  {
    /* Do not allow another Poll request before the confirm is received. */
    waitPollConfirm = TRUE;
    /* Restart timer. */
    Timer_Reset();
  }
}
```

Sending this request means that the device is asking the coordinator if there is any data avilable in the coordinator for the device. The coordinator sends back an answer in a MLME-POLL.response, resulting in a MLME-POLL.confirm on the device as shown in the following example code.

```
uint8_t App_HandleMlmeInput(nwkMessage_t *pMsg)
{
  if(pMsg == NULL)
    return errorNoMessage;

  /* Handle the incoming message. The type determines the sort of
processing.*/
  switch(pMsg->msgType) {
  case gNwkPollCnf_c:
    if(pMsg->msgData.pollCnf.status != gSuccess_c)
    {
      /* The Poll Confirm status was not successful. Usually this happens if
         no data was available at the coordinator. In this case we start
         polling at a lower rate to conserve power. */
      pollInterval = POLL_INTERVAL_SLOW;

      /* If we get to this point, then no data was available, and we
         allow a new poll request. Otherwise, we wait for the data
         indication before allowing the next poll request. */
      waitPollConfirm = FALSE;
    }
    break;
  }
```

```
  return errorNoError;
}
```

As this code example shows, a `status` parameter different from `gSuccess_c` indicates that there is no data available for the device on the coordinator. In this example, the device uses a timer for polling the coordinator at regular intervals. If the coordinator does not have any data, the polling rate is decreased until the coordinator starts transmitting data again and then the polling interval is again increased. Using this polling scheme conserves power in the device (the transceiver is on less often) but other applications may want to use another polling scheme and may choose to poll differently.
Should the coordinator send back a MLME-POLL.response that indicates that it has data queued for the device, the coordinator will, immediately after the MLME-POLL.response, send its data to the device, which in turn will receive the data packet on the MCPS interface. The application does not need to take any action on a MLME-POLL.confirm message with `status==gSuccess_c`). This extends the `App_HandleMcpsInput()` function on the device that as shown earlier in this chapter. See the `MyApp_Ex06b.c` file for more details.

```
void App_HandleMcpsInput(mcpsToNwkMessage_t *pMsgIn)
{
  switch(pMsgIn->msgType)
  {
    /* The MCPS-Data confirm is sent by the MAC to the network
       or application layer when data has been sent. */
  case gMcpsDataCnf_c:
    if(numPendingPackets)
      numPendingPackets--;
    break;

  case gMcpsDataInd_c:
    /* Copy the received data to the UART. */
    Uart_Tx(pMsgIn->msgData.dataInd.msdu, pMsgIn->msgData.dataInd.msduLength);
    /* Since we received data, the coordinator might have more to send. We
       reduce the polling interval to raise the throughput while data is
       available. */
    pollInterval = POLL_INTERVAL_FAST;
    /* Allow another MLME-Poll request. */
    waitPollConfirm = FALSE;
    break;
  }
}
```

As was the case for the coordinator, the device simply passes the data to the UART and completes the transparent UART example.

3.6 Summary

Users should now be able to understand the basic steps in creating and managing a simple, non-beacon IEEE 802.15.4 PAN.

1. Starting and initializing the Freescale IEEE 802.15.4 MAC software.

2. Performing and energy detection (ED) scan.

3. Starting a non-beacon PAN.
4. Performaing an active scan.
5. Associating a device and a coordinator.
6. Performing a direct and indirect data transfer.

When writing their own application, users may choose to employ a less complex data transfer model. That is, only one outstanding data request at a time or use a different polling scheme. The design depends upon what the application is supposed to do and the performance criteria imposed on the application. Items to consider are code size, power consumption, throughput, among others. This example is intended to introduce just the basic concepts of a non-beacon IEEE 802.15.4 PAN though there are many other ways of achieving the same functionality, the basic concepts remain the same.