Kinetis Simple Media Access Controller (SMAC)

Reference Manual

Document Number: KSMACRM
Rev. 0
01/2017

About This Book

This manual provides a detailed description of the Kinetis Simple Media Access Controller (SMAC). This software is designed for use specifically with the MKW2xD and MCR20A platforms.

The MKW2xD is a highly-integrated, cost-effective System in Package (SiP), 2.4 GHz wireless node solution with an OQPSK modulation-capable transceiver and low-power, ARM® Cortex®-M4 32-bit microcontroller.

The MCR20AVHM transceiver is a low power, high-performance 2.4 GHz, IEEE 802.15.4-compliant transceiver with connectivity to a broad range of microcontrollers, including the Kinetis family of products. To support the MCR20A transceiver, NXP provides a NXP Freedom Development platform (FRDM-CR20A).

The Kinetis SMAC software is pre-defined to operate in the 2.4 GHz band with OQPSK modulation, over an IEEE® 802.15.4 compliant PHY layer.

Audience

This document is intended for application developers working on custom wireless applications that employ the aforementioned devices. The latest version of the NXP Kinetis SMAC is available on the NXP website.

Organization

This document is organized into three chapters.

- Chapter 1, "Kinetis SMAC introduction" — this chapter introduces Kinetis SMAC features and functionality.
- Chapter 2, "Software architecture" — this chapter describes Kinetis SMAC software architecture.
- Chapter 3, "Primitives" — this chapter provides a detailed description of Kinetis SMAC primitives.

Revision History

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

Revision History

Rev. number	Date	Substantive changes
0	01/2017	Initial release.

Conventions

This document uses the following notational conventions:

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

- All source code examples are in C.

Definitions, Acronyms, and Abbreviations

The following list defines the acronyms and abbreviations used in this document.

Kinetis	Kinetis Wireless Platforms (FRDM-KW24, USB-KW24D512, FRDM-K64F with FRDM-CR20A and FRDM-KL46Z with FRDM-CR20A)
GUI	Graphical User Interface
MAC	Medium Access Control
MCU	MicroController Unit
NVM	Non-Volatile Memory
PC	Personal Computer
TERM	Serial Port Terminal Application
XCVR	Transceiver
PCB	Printed Circuit Board
OTA	Over the air.
SAP	Service Access Point
ACK	Acknowledge
AA	Automatic ACK
LBT	Listen Before Talk
RX	Receive(r)
TX	Transmit(ter)
CCA	Clear Channel Assessment
ED	Energy Detect
RTOS	Real Time Operating System

References

The following sources were referenced to produce this book:

1. *NXP MKW2xD Reference Manual* (document MKW2xDxxxRM)
2. *NXP MCR20A Reference Manual* (document MCR20RM)

Chapter 1
Kinetis SMAC introduction

The NXP Kinetis Simple Media Access Controller (SMAC) is a simple ANSI C-based codebase available as sample source code. It is used for developing proprietary RF transceiver applications using NXP's MKW2xD 2.4 GHz transceiver plus microcontroller and MCR20A transceiver attached (but not limited) to either a MK64F12 or MKL46Z4 microcontroller.

The MKW2xD is a system-in-package (SiP) device that includes an ARM Cortex-M4-based microcontroller and 2.4 ISM band radio front-end device. Features of the MKW2xD include:

- MCU has a 32-bit ARM Cortex-M4 CPU with a full set of peripheral functions.
- MCU has up to 512 KB flash and 64 KB SRAM (MKW21D256 has 256 KB of flash, and 32 KB of RAM).
- Full featured, programmable transceiver that supports the OQPSK modulation scheme.
- The MKW2xD has internal connections between the MCU and transceiver:
  - The MCU communicates with the transceiver through SPI and GPIO pins.

The MCR20A transceiver is a 2.4 GHz Industrial, Scientific, and Medical (ISM), and Medical Body Area Network (MBAN) transceiver intended for the IEEE® 802.15.4 Standard. The MCR20A device is a standalone transceiver that is normally combined with a software stack and a NXP Kinetis K series, M series, or other microcontroller (MCU) to implement an IEEE 802.15.4 Standard platform solution.

Features of the MCR20A include:

- Fully compliant IEEE 802.15.4 Standard 2006 transceiver supports 250 kbit/s OQPSK data in 5.0 MHz channels and full spread-spectrum encode and decode. Also extends radio operation to the 2.36 GHz to 2.40 GHz Medical Band (MBAN) frequencies with IEEE 802.15.4j channel, spacing, and modulation requirements.
- 2.4 GHz frequency band of operation (ISM).
- 250 kbit/s data rate with O-QPSK modulation in 5.0 MHz channels with direct sequence spread spectrum (DSSS) encode and decode.
- Operates on one of 16 selectable ISM channels per IEEE 802.15.4 specification.
- Programmable output power.
- Supports 2.36 GHz to 2.40 GHz Medical Band (MBAN) frequencies with IEEE 802.15.4j channel, spacing, and modulation requirements.
- Hardware acceleration for IEEE® 802.15.4 Standard.
- Complete 802.15.4 onboard modem.
- IEEE 802.15.4 Standard 2006 packet processor/sequencer with receiver frame filtering.
- Random number generator.

- Support for dual PAN ID mode.
- Internal event timer block with four comparators to assist sequencer and provide timer capability.
- The MCR20A has external connections with the MCU:
  - The transceiver communicates with a target MCU through SPI, eight (8) software programmable GPIO pins, and an output interrupt pin.

NOTE

It is highly recommended the SMAC user be familiar with the supported devices. Additional details can be found in the devices data sheets (MKW2xDXXX and MCR20AVHM) and *Reference Manuals* (MKW2xDxxxRM and MCR20RM).

The Kinetis SMAC is a small codebase that provides simple communication and test applications based on drivers, an 802.15.4 compliant PHY, and framework utilities available as source code. This environment is useful for hardware and RF debug, hardware standards certification, and developing proprietary applications. The SMAC is provided as part of the Connectivity Software Package available for the aforementioned devices.

To use any of the existing applications available in the software package, users must download and open the available demo applications in the corresponding development environment (IDE).

SMAC features include:

- Compact footprint:
  - Between 2 to 3 KB of flash required, depending on configuration used.
  - Less than 500 bytes RAM, depending on configuration used.
- Low-power, proprietary, bidirectional RF communication link.
- Both radios allow packet filtering by hardware, checking the preamble and the synchronization word, which reduces software overhead and memory footprint.
- Broadcast communication.
- Unicast communication — SMAC includes a Node Address 16-bit field. This allows SMAC to perform unicast transmissions. To change the address of a node, modify the `gNodeAddress_c` constant inside the `SMAC_Interface.h` file, or call `SMACSetShortSrcAddress(uint16_t nwShortAddress)`. The address is set to 0xBEAD by default. Some of the demo applications allow you to change this address at runtime.
- Change of current PAN. The SMAC packet uses a short 802.15.4 compliant header with a hard-coded configuration for frame control which allows the user to switch between PANs. The PAN address has also 16 bits (`gDefaultPanID_c`). This address can be modified both by changing the default value from `SMAC_Interface.h` file or by calling `SMACSetPanID(uint16_t nwShortPanID.`
- No blocking functions within the SMAC.
- Easy-to-use sample applications included.
- Light-weight, custom LBT algorithm.
- Light-weight, custom AA mechanism, which is transparent to the user after enabling the feature.

- Encryption using Advanced Encryption Standard in Cipher Block Chaining mode, with configurable initial vector and key.
- Configurable number of retries and backoff interval.
- Inter-layer communication using SAPs.
- SMAC also filters packets that have correct addressing information (pass address filtering) but are not in the expected format (short addressing, no security, data frame).

1.1 Kinetis SMAC-based demonstration applications

The following is a list of Kinetis SMAC-based demonstration applications:

- PC-based Connectivity Test Application which requires a TERM. This application allows the user to perform basic communication tests and several advanced XCVR tests.
- PC-based Wireless Messenger Application which requires a TERM and is presented in the form of a messenger-like application. This demo application highlights the “Listen Before Talk” and “Automatic ACK” mechanisms, allowing the user to enable, disable and configure them at runtime.
- PC-based Wireless UART Application which requires either a TERM or an application capable of reading/writing from/to a serial port. This application is used as a wireless UART bridge between two or more (one to many) platforms. It can be configured to use the previously mentioned mechanisms, but the configuration must be done at compile time.
- PC-based Low Power Demo Application which requires a TERM. This application helps the user to learn how to enable low power modes on the MKW2xD, MCR20A, MK64F12 and MKL46Z4 platforms. It also contains a scenario based on the Very Low Power Stop (VPLS) mode and SMAC to demonstrate how a low power mode can be used in a connectivity stack.

1.2 Platform requirements

SMAC can be used with any customer target application or board. However, NXP provides several development platform (Freedom board and USB dongle) designs with LEDs, push-buttons, and other modules included.

1.3 MCU resources used by SMAC

SMAC does not use MCU resources directly. All accesses to the MCU resources are performed using the framework, drivers and PHY.

1.4 SMAC basic initialization

Before transmitting, receiving, or performing any other SMAC operation described in this manual, the system protocol must be initialized to configure the transceiver with correct functional settings and to set SMAC's state machine to known states. To initialize the SMAC protocol, perform the following tasks in order:

1. Initialize MCU interrupts and peripherals. This initialization is included in every demo in the hardware_init(void) function, available as source code.

   — Initialize the LED, Keyboard, Serial Manager, Timers Manager, Memory Manager, and drivers depending on application needs.

   ```
   MEM_Init();
   TMR_Init();
   LED_Init();
   SerialManager_Init();
   ```

   — Initalize PHY layer.

   ```
   Phy_Init();
   ```

2. Initialize SMAC, to set the SMAC state machine to default, configure addressing with default values, initialize the RNG used for the first sequence number value, and the random backoff.

   ```
   InitSmac();
   ```

3. Set the SAP handlers so that SMAC can notify the application on asynchronous events on both data and management layers.

   ```
   void Smac_RegisterSapHandlers(
                                 SMAC_APP_MCPS_SapHandler_t pSMAC_APP_MCPS_SapHandler,
                                 SMAC_APP_MLME_SapHandler_t pSMAC_APP_MLME_SapHandler,
                                 instanceId_t smacInstanceId
                                   )
   ```

4. Reserve the RAM memory space needed by SMAC to allocate the received and transmitted OTA messages by declaring the buffers that must be of the size of maximum payload (SDU) added to the size of the structure holding the packet:

   ```
   uint8_t RxDataBuffer[gMaxSmacSDULength_c  + sizeof(rxPacket_t)];
   rxPacket_t *RxPacket;

   uint8_t TxDataBuffer[gMaxSmacSDULength_c  + sizeof(txPacket_t)];
   txPacket_t *TxPacket;

   RxPacket = (rxPacket_t*)RxDataBuffer;
   TxPacket = (txPacket_t*)TxDataBuffer;
   ```

Chapter 2
Software architecture

This chapter describes the SMAC software architecture. All of the SMAC source code is always included in the application. SMAC is primarily a set of utility functions or building blocks that users can use to build simple communication applications.

2.1 Block diagram

Figure 2-1 shows a simplified SMAC based stack block diagram.

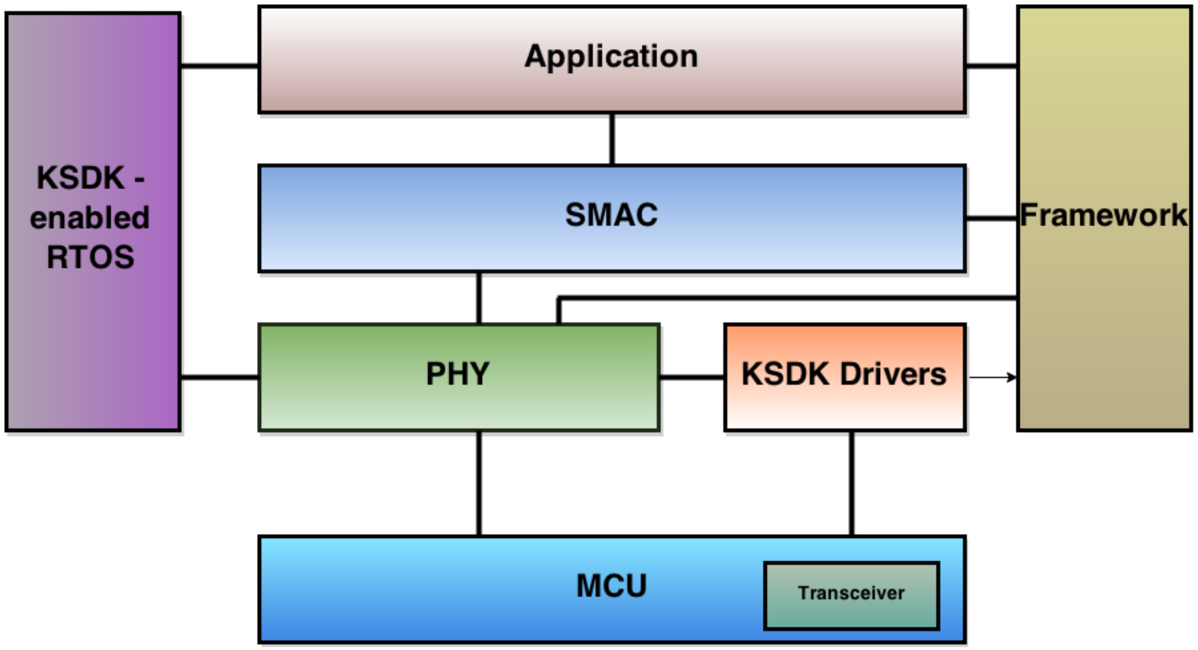

Figure 2-1. SMAC System Decomposition

An application programming interface (API) is implemented in the SMAC as a C header file (`.h`) that allows access to the code. The code includes the API to specific functions. Thus, the application interface with the SMAC is accomplished by including the `SMAC_Interface.h` file, which makes reference to the required functions within the SMAC and provides the application with desired functionality.

NOTE

Kinetis SMAC demo projects support only the target boards designated in the files.

2.2 SMAC data types and structures

The SMAC fundamental data types and defined structures are discussed in the following sections.

2.2.1 Fundamental data types

The following list shows the fundamental data types and the naming convention used in the SMAC implementation:

uint8_t	Unsigned 8-bit definition
uint16_t	Unsigned 16-bit definition
uint32_t	Unsigned 32-bit definition
int8_t	Signed 8-bit definition
int16_t	Signed 16-bit definition
int32_t	Signed 32-bit definition

These data types are used in the SMAC project as well as in the applications projects. They are defined in the `EmbeddedTypes.h` file.

2.2.2 rxPacket_t

This structure defines the variable used for SMAC received data buffer:

```
typedef struct rxPacket_tag{
  uint8_t    u8MaxDataLength;
  rxStatus_t rxStatus;
  uint8_t    u8DataLength;
  smacHeader_t smacHeader;
  smacPdu_t  smacPdu;
}rxPacket_t;
```

Members

u8MaxDataLength	Max number of bytes to be received.
rxStatus	Indicates the reception state. See `rxStatus_t` data type for more detail.
u8DataLength	Number of received bytes.
smacPdu	The SMAC protocol data unit.
smacHeader	SMAC structure that defines the header used. NXP recommends that the user does not modify this structure directly, but through the associated functions.

Usage

This data type is used by an application in the following manner:

1. Declare a buffer to store a packet to be received OTA. NXP recommends the size of this buffer to be at least as long as the biggest packet to be received by the application.
2. Declare a pointer of the type rxPacket_t.
3. Initialize the pointer to point to the buffer declared at the first step.
4. Initialize the `u8MaxDataLength` member of the packet structure. The SMAC will filter all the received packets having a payload size bigger than `u8MaxDataLength`.
5. Use the pointer as the argument when calling `MLMERXEnableRequest`:

```
uint8_t RxDataBuffer[gMaxSmacSDULength_c + sizeof(rxPacket_t)];
rxPacket_t *RxPacket;
RxPacket = (rxPacket_t*)RxDataBuffer;
RxPacket->u8MaxDataLength = gMaxSmacSDULength_c;
RxEnableResult = MLMERXEnableRequest(RxPacket, 0);
```

You can use a variable of the type `smacErrors_t` to store the result of executing `MLMERXEnableRequest` function.

2.2.3 smacHeader_t

This structure defines the variable used for the SMAC header:

```
typedef PACKED_STRUCT smacHeader_tag{
  uint16_t    frameControl;
  uint8_t     seqNo;
  uint16_t    panId;
  uint16_t    destAddr;
  uint16_t    srcAddr;
}smacHeader_t;
```

Members

frameControl	Frame control configuration. The value is set each time SMACFillHeader is called and should not be changed.
seqNo	The Sequence number is updated each time a data request is performed.
panId	The value of the source and destination PAN address. It is recommended to be changed through the associated function.
destAddr	The short destination address.
srcAddr	The short source address.

Usage

NXP recommends that the user does not access this structure directly but through the associated functions.

2.2.4 rxStatus_t

This enumeration lists all the possible reception states:

```
typedef enum rxStatus_tag
{
  rxInitStatus,
  rxProcessingReceptionStatus_c,
  rxSuccessStatus_c,
  rxTimeOutStatus_c,
  rxAbortedStatus_c,
  rxMaxStatus_c
} rxStatus_t;
```

Members

rxInitStatus	The RTOS based SMAC does not use this.
rxProcessingReceptionStatus_c	This state is set when the SMAC is in the middle of receiving a packet.
rxSuccessStatus_c	This is one of the possible finish conditions for a received packet that was successfully received and could be checked by the indication functions.
rxTimeOutStatus_c	This is another of the possible finish conditions for a timeout condition and could be checked by the indication functions.
rxAbortedStatus_c	This status is set when SMAC drops a packet (on SMAC specific criteria) validated by PHY and the enter reception request was performed with a nonzero timeout.
rxMaxStatus_c	This element indicates the total number of possible reception states.

2.2.5 smacPdu_t

This type defines the SMAC's basic protocol data unit:

```
typedef struct smacPdu_tag{
      uint8_t smacPdu[1];
}smacPdu_t;
```

Members

smacPdu[1].	Starting position of the buffer where TX or RX data is stored.

2.2.6 txPacket_t

This structure defines the type of variable to be transmitted by the SMAC. It is located in the `SMAC_Interface.h` file and is defined as follows:

```
typedef struct txPacket_tag
{
  uint8_t u8DataLength;
  smacHeader_t smacHeader;
  smacPdu_t smacPdu;
}txPacket_t;
```

Members

u8DataLength	The number of bytes to transmit.
smacHeader	SMAC structure that defines the header used. NXP recommends that the user does not modify this structure directly, but through the associated functions.
smacPdu	The SMAC protocol data unit.

Usage

This data type is used by an application in the following manner:

1. Declare a buffer to store the packet to be transmitted OTA. NXP recommends the size of this buffer is at least as long as the biggest packet to be transmitted by the application.
2. Declare a pointer of the type txPacket_t.
3. Initialize the pointer to point to the buffer declared at the first step.
4. Copy the desired data into the payload.
5. Set u8DataLength to the size (in bytes) of the payload.
6. Use the pointer as the argument when calling MCPSDataRequest.

```
uint8_t TxDataBuffer[gMaxSmacSDULength_c + sizeof(txPacket_t)];
txPacket_t *TxPacket;
...
TxPacket = (txPacket_t*)TxDataBuffer;
FLib_MemCpy(TxPacket->smacPdu.smacPdu, dataToBeSentBuffer, payloadSizeBytes);
TxPacket->u8DataLength = payloadSizeBytes;
DataRequestResult = MCPSDataRequest(TxPacket);
```

You can use a variable of the type smacErrors_t to store the result of executing MCPSDataRequest function.

2.2.7 channels_t

Definition for RF channels. The number of channel varies in each defined operating band for sub-1 GHz stacks, but it is fixed for the 2.4 GHz. First logical channel in all bands is 0 for sub-1GHz and 11 for 2.4 GHz. It is defined as follows:

```
typedef enum channels_tag
{
#include "SMAC_Channels.h"
} channels_t;
```

Each application derives the minimum and maximum channel values from the enumeration above. SMAC only keeps an enumeration of all the possible channel numbers.

Members

None

2.2.8 smacErrors_t

This enumeration is used as the set of possible return values on most of the SMAC API functions and is located in the `SMAC_Interface.h`. Also, some of the messages sent by SMAC to the application use this enumeration as a status.

```
typedef enum smacErrors_tag{
  gErrorNoError_c = 0,
  gErrorBusy_c,
  gErrorChannelBusy_c,
  gErrorNoAck_c,
  gErrorOutOfRange_c,
  gErrorNoResourcesAvailable_c,
  gErrorNoValidCondition_c,
  gErrorCorrupted_c,
  gErrorMaxError_c
} smacErrors_t;
```

Members

gErrorNoError_c	SMAC accepts the request and processes it. This return value does not necessarily mean that the action requested was successfully executed. It means only that it was accepted for processing. This value is also used as a return status in the SMAC to application SAPs. For example, if a packet has been succesfully sent, the message will have a data confirm field with this status. Also, this value is returned in the CCA confirm message if the scanned channel is found idle.
gErrorBusy_c	This constant is returned when the SMAC layer is not in an idle state, and it cannot perform the requested action.
gErrorChannelBusy_c	The custom “Listen Before Talk” algorithm detected a busy channel more times than the configured number of retries. Also, a CCA confirm message can have this value if the channel is found busy.
gErrorNoAck_c	The custom “Automatic Ack” mechanism detected that no acknowledgement packet has been received more times than the configured number of retries.
gErrorOutOfRange_c	A certain parameter configured by the application is not in the valid range.
gErrorNoValidCondition_c	Returned when requesting an action on an invalid environment. Requesting SMAC operations when it has not been initialized or requesting to disable RX when SMAC was not in a receiving or idle state, or setting a number of retries without enabling the “LBT and AA” features.
gErrorCorrupted_c	Not implemented in the RTOS based SMAC.
gErrorMaxError_c	This constant indicates the total number of returned constants.

2.2.9 smacMultiPanInstances_t

This enumeration is used as a list of identifiers of the PAN's supported simultaneously by SMAC.

```
typedef enum smacMultiPanInstances_tag
{
  gSmacPan0_c = 0,
#if gMpmMaxPANs_c == 2
  gSmacPan1_c,
#endif
  gSmacMaxPan_c
}smacMultiPanInstances_t;
```

Members

gSmacPan0_c	The first PAN identifier used inside the dual PAN API.
gSmacPan1_c	This identifier exists only if gMpmMaxPANs_c is globally defined as 2 and it is the identifier of the alternate PAN.
gSmacMaxPan_c	This member shows the maximum number of PAN's simultaneously handled by SMAC.

2.2.10 txContextConfig_t

```
typedef struct txContextConfig_tag
{
  bool_t ccaBeforeTx;
  bool_t autoAck;
  uint8_t retryCountCCAFail;
  uint8_t retryCountAckFail;
}txContextConfig_t;
```

Members

ccaBeforeTx	`bool_t` value to enable/disable the “LBT” mechanism.
autoAck	`bool_t` value to enable/disable the “AA” mechanism.
retryCountCCAFail	This value specifies the number of times the SMAC will attempt to retransmit a packet if “LBT” is enabled and channel is found busy.
retryCountAckFail	This value specifies the number of times the SMAC will attempt to retransmit a packet if “AA” is enabled and no acknowledgement message is received in the expected time frame.

2.2.11 smacTestMode_t

```
typedef enum smacTestMode_tag
{
  gTestModeForceIdle_c = 0,
  gTestModeContinuousTxModulated_c,
  gTestModeContinuousTxUnmodulated_c,
  gTestModePRBS9_c,
  gTestModeContinuousRxBER_c,
  gMaxTestMode_c
} smacTestMode_t;
```

This enumeration is used only in the Connectivity Test Application, to select the type of test to be performed. Keep in mind that all the decisions are taken at application level and this enumeration is used only as a reference for designing the test modes.

2.2.12 smacEncryptionKeyIV_t

```
typedef struct smacEncryptionKeyIV_tag
{
  uint8_t IV[16];
  uint8_t KEY[16];
}smacEncryptionKeyIV_t;
```

Members

IV	The initial vector used by the CBC mode of AES.
KEY	The encryption / decryption key used by the CBC mode of AES.

Usage

This data type is used internally by SMAC. Call SMAC_SetIVKey with two 16 byte buffer pointers as parameters to change the SMAC initial vector and encryption key settings.

2.3 SMAC to application messaging

The RTOS based SMAC communicates with the application layer in two ways:

- Directly, through the return value of the functions, if the request is synchronous (change channel, output power, etc)
- Indirectly, through SAPs for asynchronous events (data confirm, ED/CCA confirm, data indication, timeout indication).

Both SAPs (data and management) pass information to the application using a messaging system. The data structures used by this system are described below.

```
typedef enum smacMessageDefs_tag
{
  gMcpsDataCnf_c,
  gMcpsDataInd_c,

  gMlmeCcaCnf_c,

  gMlmeEdCnf_c,


  gMlmeTimeoutInd_c,


  gMlme_UnexpectedRadioResetInd_c,
}smacMessageDefs_t;
```

The above enumeration summarizes the types of messages passed through SAPs. As mentioned earlier, there are data confirm, data indication (data layer), CCA confirm, ED confirm, timeout indication, and unexpected radio reset indication (management layer) messages. Each message type is accompanied by corresponding message data. The main structures that build the message data are described below.

Table 2-1. Message types and associated data structures

Index	Message type	Associated data structures	Description
1	`gMcpsDataCnf_c`	smacDataCnf_t	Contains a *smacErrors_t* element. See *Section 2.2.8, "smacErrors_t"*
2	`gMcpsDataInd_c`	smacDataInd_t	*u8LastRxRssi* value indicating the average RSSI obtained during the reception *pRxPacket* pointer to the packet passed as parameter to *MLMERXEnableRequest*.
3	`gMlmeCcaCnf_c`	smacCcaCnf_t	Contains a *smacErrors_t* element. See *Section 2.2.8, "smacErrors_t"*
4	`gMlmeEdCnf_c`	smacEdCnf_t	*status* This is a *smacErrors_t* element. If PHY succesfully performs the ED it's value will be *gErrorsNoError_c* *energyLevel* The value of the energy level register. *energyLeveldB* The value of the energy level converted to dBm. *scannedChannel* The channel number of the scanned channel.
5	`gMlmeTimeoutInd_c`	*none*	—
6	`gMlme_UnexpectedRadioResetInd_c`	*none*	—

All taken into consideration, the two types of messages used by the SMAC to application SAPs have the following form:

```
typedef  struct smacToAppMlmeMessage_tag
{
  smacMessageDefs_t           msgType;
  uint8_t                     appInstanceId;
  union
  {
    smacCcaCnf_t            ccaCnf;
    smacEdCnf_t            edCnf;
  }msgData;
} smacToAppMlmeMessage_t;
```

```
typedef  struct smacToAppDataMessage_tag
{
  smacMessageDefs_t          msgType;
  uint8_t                    appInstanceId;
  union
  {
    smacDataCnf_t             dataCnf;
    smacDataInd_t             dataInd;
  }msgData;
} smacToAppDataMessage_t;
```

The SMAC to application SAP handlers are function pointers of a special type. When application specifies the functions to handle asynchronous responses, the SAP handlers aquire the value of those functions. The definitions of the handlers are as follows:

```
typedef smacErrors_t ( * SMAC_APP_MCPS_SapHandler_t)(smacToAppDataMessage_t * pMsg,
instanceId_t instanceId);

typedef smacErrors_t ( * SMAC_APP_MLME_SapHandler_t)(smacToAppMlmeMessage_t * pMsg,
instanceId_t instanceId);
```

Chapter 3
Primitives

The following sections provide a detailed description of SMAC primitives associated with the API.

3.1 MCPSDataRequest

This data primitive is used to send an over-the-air (OTA) packet. This is an asynchronous function, which means it asks SMAC to transmit an OTA packet, but transmission could continue after the function returns.

Prototype

```
smacErrors_t MCPSDataRequest(txPacket_t *psTxPacket);
```

Arguments

txPacket_t *psTxPacket	Pointer to the packet to be transmitted.

Returns

gErrorNoError_c	Everything is OK, and the transmission will be performed.
gErrorOutOfRange_c	One of the members in the pTxMessage structure is out of range (invalid buffer size or data buffer pointer is NULL).
gErrorBusy_c	The radio is performing another action and could not attend this request.
gErrorNoValidCondition_c	The SMAC layer has not been initialized.
gErrorNoResourcesAvailable_c	The PHY cannot process an SMAC request, so SMAC cannot process it, or the memory manager is unable to allocate another buffer.

Usage

- SMAC must be initialized before calling this function.
- Declare a variable of the type `smacErrors_t` to save the result of the function execution.
- Prepare the `txPacket_t` parameter as explained in Section 2.2.6, "txPacket_t" declaration and usage.
- Call the MCPSDataRequest function.
- If the function call result is different than gErrorNoError_c, the application should handle the error returned. For instance, if the result is `gErrorBusy_c`, the application should wait for the radio to finish a previous operation.

```
    uint8_t TxDataBuffer[gMaxSmacSDULength_c + sizeof(txPacket_t)];
```

```
          txPacket_t *TxPacket;
          smacErrors_t smacError;
          ...
          TxPacket = (txPacket_t*)TxDataBuffer;
          TxPacket->u8DataLength = payloadLength;
          //Copy the data to send into the smacPdu of the packet
          FLib_MemCpy(TxPacket->smacPdu.smacPdu, bufferToSend, payloadLength);
          smacError = MCPSDataRequest(TxPacket);
          ...
```

Implementation

This `MCPSDataRequest` primitive creates a message for the PHY task and fills it in respect to the user configurations prior to this call and to the information contained in the packet.

3.2 MLMETXDisableRequest

This function places the radio into standby, and places the PHY and SMAC state machines into idle, if current operation is TX. It does not explicitly check if SMAC is in a transmitting state, but it clears the SMAC buffer containing the packet to be sent, which makes it ideal for using when application wants to switch from TX to idle.

Prototype

```
void MLMETXDisableRequest(void);
```

Arguments

None.

Returns

None. The function will forcibly set the transceiver to standby, and put the PHY and SMAC state machines into idle, so no return value is needed.

Usage

Call `MLMETXDisableRequest()`.

Implementation

This primitive creates a message for PHY, and sets message type as set transceiver state request, with value of force transceiver off. After passing the message to PHY, SMAC checks if a TX is in progress and clears the buffer containing the packet.

3.3 MLMEConfigureTxContext

This function aids the user in enabling/disabling the “LBT” and “AA” mechanisms and also to configure the number of retries in case channel is found busy or no acknowledgement message is received.

Prototype

```
smacErrors_t MLMEConfigureTxContext(txContextConfig_t* pTxConfig);
```

Arguments

txContextConfig_t* pTxConfig: pointer to a configuration structure containing the information described above.

Returns

gErrorNoError_c	The desired configuration is applied succesfully.
gErrorNoValidCondition_c	The number of retries is set but the enable field is set to FALSE.
gErrorOutOfRange_c	The number of retries exceeds `gMaxRetriesAllowed_c`.

Usage

- Declare a structure of `txContextConfig_t` type.
- Set the desired values to the members.
- Call `MLMEConfigureTxContext` with the address of the declared structure as parameter.
- Capture the return value in a smacErrors_t variable and handle the result.

```
txContextConfig_t txConfigContext;
txConfigContext.autoAck           = TRUE; //”AA” is enabled
txConfigContext.ccaBeforeTx       = FALSE; //”LBT” is disabled
txConfigContext.retryCountAckFail = 0;// no retries in case no ACK is received
txConfigContext.retryCountCCAFail = 0;// no retries in case of channel busy

smacErrors_t err = MLMEConfigureTxContext(&txConfigContext);

...
```

Implementation

This primitive configures the way SMAC will handle data requests and responses from PHY according to the parameters described by the `txContextConfig_t structure`. Also, requests forwarded by SMAC to PHY depend on addressing and `txContextConfig_t` information.

3.4 MLMERXEnableRequest

This function places the radio into receive mode on the channel preselected by `MLMESetChannelRequest()`.

Prototype

```
smacErrors_t MLMERXEnableRequest(rxPacket_t *gsRxPacket, uint32_t u32Timeout);
```

Arguments

rxPacket_t *gsRxPacket: Pointer to the structure where the reception results will be stored.

uint32_t u32Timeout: 32-bit timeout value in symbol duration. One symbol duration is equivalent to a 16 µs duration.

Returns

gErrorNoError_c	Everything is OK, and the reception will be performed
gErrorOutOfRange_c	One of the members in the `rxPacket_t` structure is out of range (not valid buffer size or data buffer pointer is NULL
gErrorBusy_c	The radio is performing another action and could not attend this request.
gErrorNoValidCondition_c	The SMAC has not been initialized.
gErrorNoResourcesAvailable_c	The PHY cannot process a SMAC request, so the SMAC cannot process it.

Usage

- SMAC must be initialized before calling this function.
- Declare a variable of the type `smacErrors_t` to save the result of the function execution.
- Prepare the `rxPacket_t` parameter as explained in Section 2.2.2, “rxPacket_t” declaration and usage.
- Call MLMERXEnableRequest function.
- If the result of the call of the function is different to `gErrorNoError_c`, the application may handle the error returned. For instance, if the result is `gErrorBusy_c`, the application should wait for the radio to finish a previous operation.

```
uint8_t RxDataBuffer[gMaxSmacSDULength_c + sizeof(rxPacket_t)];
rxPacket_t *RxPacket;
smacErrors_t smacError;

RxPacket = (rxPacket_t*)RxDataBuffer;
RxPacket->u8MaxDataLength = gMaxSmacSDULength_c;
smacError = MLMERXEnableRequest(RxPacket, 0);

...
```

NOTE

- The return of anything different than `gErrorNoError_c` implies that the receiver did not go into receive mode.
- 32-bit timeout value of zero causes the receiver to never timeout and stay in receive mode until a valid data packet is received or the `MLMERXDisableRequest` function is called.
- To turn off the receiver before a valid packet is received, the `MLMERXDisableRequest` call can be used.

- If timeout is not zero and a valid packet with length greater than `u8MaxDataLength` is received, SMAC will send a data indication message and will set `rxAbortedStatus_c` in the `rxStatus_t` field of the `rxPacket_t` variable.
- When using security, although the maximum allowed payload for transmission is `gMaxSmacSDULength_c`, for reception, the user should configure the `u8MaxDataLength` field to `gMaxSmacSDULength_c` + 16 (maximum number of padding bytes for the encryption algorithm) so that SMAC will not filter out received packets with `gMaxSmacSDULength_c` size.

Implementation

This primitive creates a message for PHY, completes the message with the appropriate values and fills the timeout field with the value passed through the timeout parameter. If this value is 0, SMAC will create a set PIB request, asking PHY to enable the *gPhyPibRxOnWhenIdle* attribute.

3.5 MLMERXDisableRequest

Returns the radio to idle mode from receive mode.

Prototype

```
smacErrors_t MLMERXDisableRequest(void);
```

Arguments

None.

Returns

gErrorNoError_c	The request was processed and the transceiver is in idle.
gErrorNoValidCondition_c	The radio is not in RX state, or SMAC is not initialized.
gErrorBusy_c	The radio is performing another action and could not attend this request.

Usage

```
Call MLMERXDisableRequest ()
```

NOTE

This function can be used to turn off the receiver before a timeout occurs or when the receiver is in the always-on mode.

Implementation

This function creates a message for PHY and if the timeout value from MLMERXEnableRequest was 0, the message is filled as a set PIB request, requiring the *gPhyPibRxOnWhenIdle* to be set to 0. If the timeout value is greater than 0, the message is filled as a set transceiver state request, disabling the receiver.

It aborts the current requested action, puts the PHY in the idle state, and sets the transceiver in standby mode. It also disables any previous timeout programmed.

3.6 MLMELinkQuality

This function returns an integer value that is link quality value from the last received packet, offering information on how good is the "link" between the transmitter and the receiver. The LQI value is between 0 and 255, where 0 means bad "link" and 255 is the exact opposite.

Prototype

```
uint8_t MLMELinkQuality(void);
```

Arguments

None.

Returns

uint8_t	8-bit value representing the link quality value. Returns the result in `smacLastDataRxParams.linkQuality`.
Zero	SMAC has not been initialized.

Usage

Call the `MLMELinkQuality()`.

Implementation

This function reads the stored value in `smacLastDataRxParams.linkQuality`. This element contains the LQI value calculated by the transceiver and interpreted by the PHY layer during the last reception.

3.7 MLMESetChannelRequest

This sets the frequency on which the radio will transmit or receive.

Prototype

```
smacErrors_t MLMESetChannelRequest(channels_t newChannel);
```

Arguments

channels_t newChannel	An 8-bit value that represents the requested channel.

Returns

gErrorNoError_c	The channel set has been performed.
gErrorBusy_c	SMAC is busy in other radio activity like transmitting/receiving data or performing a channel scan.

gErrorOutOfRange_c	The requested channel is not valid.
gErrorNoValidCondition_c	SMAC is not initialized.

Usage

Call the function `MLMESetChannelRequest(newChannel);`

NOTE

Be sure to enter a valid channel between 11 and 26.

3.8 MLMEGetChannelRequest

This function returns the current channel.

Prototype

```
channels_t MLMEGetChannelRequest(void);
```

Arguments

None.

Returns

channels_t (uint8_t)	The current RF channel.

Usage

Call `MLMEGetChannelRequest();`

3.9 MLMEPAOutputAdjust

This function adjusts the output power of the transmitter.

Prototype

```
smacErrors_t MLMEPAOutputAdjust(uint8_t u8PaValue);
```

Arguments

uint8_t u8PaValue	8-bit value for the output power desired. Values 3 – 31 are required.

Returns

gErrorOutOfRange_c	u8Power exceeds the maximum power value gMaxOutputPower_c (0x1F).
gErrorBusy_c	SMAC or PHY are busy.
gErrorNoError_c	The action is performed.

gErrorNoValidCondition_c	SMAC is not initialized.

Usage

Call `MLMEPAOutputAdjust(u8PaValue);`

NOTE

Make sure to enter a valid value for the PA output adjust.

3.10 MLMEPhySoftReset

The MLMEPhySoftReset function is called to perform a software reset to the PHY and SMAC state machines.

Prototype

```
smacErrors_t MLMEPHYSoftReset(void);
```

Arguments

None.

Returns

gErrorNoError_c	If the action is performed.
gErrorNoValidCondition_c	If SMAC is not initialized.

Usage

Call `MLMEPHYSoftReset();`

Implementation

This function creates a set transceiver state request message with force transceiver off field set and sends it to PHY.

3.11 MLMEScanRequest

This function creates an ED request message to the PHY. If the channel passed as parameter is different from the current channel, this function changes the channel before requesting the ED.

Prototype

```
smacErrors_t MLMEScanRequest(channels_t u8ChannelToScan);
```

Arguments

channels_t u8ChannelToScan: Channel to be scanned.

Returns

gErrorNoError_c	Everything is normal and the scan will be performed.
gErrorBusy_c	The radio is performing another action.
gErrorNoValidCondition_c	SMAC has not been initialized.

Usage

Call the function with the selected channel to be scanned.

```
MLMEScanRequest(u8ChannelToScan);
```

NOTE

Make sure to enter a valid channel (between 11 and 26). Switch back to the previous channel after receiving the result.

3.12 MLMECcaRequest

This function creates a CCA request message and sends it to the PHY. CCA is performed on the active channel (set with MLMESetChannelRequest). The result will be received in a message passed through the SMAC to application management SAP.

Arguments

None.

Returns

gErrorNoValidCondition_c	SMAC has not been initialized.
gErrorBusy_c	Either SMAC or PHY is busy and cannot process the request.
gErrorNoError_c	Everything is normal and the request was processed.

Usage

Call the function. The application can store the return value in a `smacErrors_t` variable and handle the error in case it occurs. For example, if the return value is gErrorBusy_c, the application can wait on this value until SMAC becomes idle.

```
smacErrors_t ReturnValue;
ReturnValue = MLMECcaRequest();
//Handle return value
...
```

Implementation

This function creates a message for PHY requesting a CCA on the currently selected channel. After passing the message through the SAP, SMAC changes it’s state to `mSmacStatePerformingCca_c`.

3.13 SMACSetShortSrcAddress

This function creates a message of set PIB request type, requesting PHY to change the short source address of the node. If the message is passed succesfully to PHY, SMAC will set it’s own source address variable to the new value, so that when SMACFillHeader is called, the updated data is filled into the header.

Arguments

uint16_t nwShortAddress: The new value of the 16 bit node address.

Returns

gErrorNoResourcesAvailable_c	The PHY layer can not handle this request.
gErrorBusy_c	PHY is busy and can not process the request.
gErrorNoError_c	Everything is normal and the request was processed.

Usage

Call the function with the desired address. The application can store the return value in a `smacErrors_t` variable and handle the error in case it occurs. For example, if the return value is `gErrorBusy_c`, the application can wait on this value until PHY becomes idle.

```
smacErrors_t ReturnValue;
ReturnValue = MLMESetShortSrcAddress(0x1234);
//Handle return value
...
```

Implementation

This function creates a message for PHY requesting to set the source address PIB to the value passed as parameter. If the request is processed, the value is also stored in the SMAC layer for fast processing in case a call to SMACFillHeader is performed.

3.14 SMACSetPanID

This function creates a message of set PIB request type, requesting PHY to change the short PAN address of the node. If the message is passed succesfully to PHY, SMAC will set it’s own PAN address variable to the new value, so that when SMACFillHeader is called, the updated data is filled into the header.

Arguments

uint16_t nwShortPanID: The new value of the 16 bit PAN address.

Returns

gErrorNoResourcesAvailable_c	The PHY layer can not handle this request.
gErrorBusy_c	PHY is busy and can not process the request.
gErrorNoError_c	Everything is normal and the request was processed.

Usage

Call the function with the desired address. The application can store the return value in a smacErrors_t variable and handle the error in case it occurs. For example, if the return value is gErrorBusy_c, the application can wait on this value until PHY becomes idle.

```
smacErrors_t ReturnValue;
ReturnValue = MLMESetShortPanID(0x0001);
//Handle return value
...
```

Implementation

This function creates a message for PHY requesting to set the PAN address PIB to the value passed as parameter. If the request is processed, the value is also stored in the SMAC layer for fast processing in case a call to SMACFillHeader is performed.

3.15 SMACFillHeader

This function has no interaction with the PHY layer. Its purpose is to aid the application in configuring the addressing for a packet to be sent. The function will fill the packet header with the updated addressing and hard-coded configuration values and will add the destination address passed as parameter.

Arguments

smacHeader_t* pSmacHeader	Pointer to the SMAC header that needs to be filled with addressing and configuration information.
uint16_t destAddr	The 16 bit destination address.

Returns

None.

Usage

Call the function if it's the first time the application uses the txPacket_t variable, or if the destination address must be changed.

```
uint8_t TxDataBuffer[gMaxSmacSDULength_c + sizeof(txPacket_t)];
txPacket_t *TxPacket;
```

```
smacErrors_t smacError;
...
TxPacket = (txPacket_t*)TxDataBuffer;
SMACFillHeader(&(TxPacket->smacHeader), gBroadcastAddress_c);
TxPacket->u8DataLength = payloadLength;
//Copy the data to send into the smacPdu of the packet
FLib_MemCpy(TxPacket->smacPdu.smacPdu, bufferToSend, payloadLength);
smacError = MCPSDataRequest(TxPacket);
...
```

Implementation

This function fills the smacHeader with default, hard-coded frame control and sequence number values. Also, it adds the addressing information (configured by calling `MLMESetShortSrcAddress` and `MLMESetPanID`) and the destination address passed as parameter.

3.16 SMAC_SetIVKey

This function sets the initial vector and encryption key for the encryption process if `gSmacUseSecurity_c` is defined.

Arguments

uint8_t* KEY	Pointer to a 16 byte buffer containing the key.
uint8_t* IV	Pointer to a 16 byte buffer containing the initial vector.

Returns

None.

Usage

Declare two buffers each with 16 byte size. Fill one of them with key information and the other with initial vector information. Call this function with pointers to the buffers as parameters.

3.17 Smac_RegisterSapHandlers

This function has no interaction with the PHY layer. Its purpose is to create a communication bridge between SMAC and application, so that SMAC can respond to asynchronous requests.

Arguments

SMAC_APP_MCPS_SapHandler_t pSMAC_APP_MCPS_SapHandle:	Pointer to the function handler for data layer response to asynchronous requests.
SMAC_APP_MLME_SapHandler_t pSMAC_APP_MLME_SapHandler:	Pointer to the function handler for management layer response to asynchronous requests (ED/CCA requests).

instanceId_t smacInstanceId:	The instance of SMAC for which the SAPs are registered. Always use 0 as value for this parameter since this version of SMAC does not support multiple instances.

Returns

None.

Usage

Implement two functions that meet the constraints of the function pointers. Then call `Smac_RegisterSapHandlers` with the names of the functions.

```
smacErrors_t smacToAppMlmeSap(smacToAppMlmeMessage_t* pMsg, instanceId_t instance)
{
  switch(pMsg->msgType)
  {
  case gMlmeEdCnf_c:
  ...
    break;
  case gMlmeCcaCnf_c:
  ...
    break;
  case gMlmeTimeoutInd_c:
  ...
    break;
  default:
    break;
  }
  MEM_BufferFree(pMsg);
  return gErrorNoError_c;
}
smacErrors_t smacToAppMcpsSap(smacToAppDataMessage_t* pMsg, instanceId_t instance)
{
  switch(pMsg->msgType)
  {
  case gMcpsDataInd_c:
    ...
    break;
  case gMcpsDataCnf_c:
    ...
    break;
  default:
    break;
  }

  MEM_BufferFree(pMsg);
  return gErrorNoError_c;
}

void InitApp
{
     ...
     Smac_RegisterSapHandlers(
                                        (SMAC_APP_MCPS_SapHandler_t)smacToAppMcpsSap,
                                        (SMAC_APP_MLME_SapHandler_t)smacToAppMlmeSap,
                                        0)
        ...
    }
```

Implementation

This function associates the SMAC internal function handlers with the ones registered by the application. Whenever an asynchronous response needs to be passed from SMAC to application, the internal handlers are called, which in turn call the ones defined by the application.

3.18 MLMESetActivePan

This function is used to change the active PAN managed by SMAC. It should be called in dual pan scenarios if the upper layer wants to make a request to SMAC on the alternate PAN.

Arguments

smacMultiPanInstances_t	panID The identifier of the PAN. Can be either 0 or 1, if gMpmMaxPANs_c is globally defined as 2, or 0 otherwise.

Returns

gErrorOutOfRange_c	The PAN identifier exceeds allowed values.
gErrorNoValidCondition_c	SMAC was not initialized.
gErrorNoError_c	The active PAN is changed. Future calls to SMAC will address the new PAN configuration.

Usage

Call this function whenever you need to start making requests to another PAN handled by SMAC.

```
smacErrors_t err = MLMESetActivePan(gSmacPan0_c);
//handle error here
If(err == gErrorNoError_c)
{
   (void)SMACSetShortSrcAddress(0xBEEF); //set node address to 0xBEEF on PAN 0.
}
err = MLMESetActivePan(gSmacPan1_c);
If(err == gErrorNoError_c)
{
   (void)SMACSetShortSrcAddress(0xCAFE); //set node address to 0xCAFE on PAN 1
}
```

Implementation

This function validates the upper layer request and if currently active PAN identifier is different from the requested PAN identifier it modifies it.

3.19 MLMEConfigureDualPanSettings

This function configures the parameters for automatically shifting between PAN's under some circumstances. For example if RX requests are addressed on both PAN's, this function can be called to modify how the PHY layer will manage them.

Arguments

bool_t bUseAutoMode	TRUE if PHY should shift between PAN's using the configured dwell time, FALSE otherwise.
bool_t bModifyDwell	TRUE if the call to this function also needs to modify the dwell time, FALSE otherwise.
uint8_t u8Prescaler	Specifies the time resolution for the dwell time. See table in MpmInterface.h for values of the prescaler.
uint8_t u8Scale	This value specifies the dwell time in respect with the above time resolution. Dwell time is Table[u8Prescaler]*(u8Scale+1).

Returns

gErrorNoValidCondition_c	This value is returned if SMAC is not initialized or dual pan feature is not enabled at compile time.
gErrorBusy_c	This value is returned if SMAC is currently servicing another request on either PAN's.
gErrorOutOfRange_c	This value is returned if bModifyDwell is TRUE but either u8Prescaler or u8Scale parameter is out of range.
gErrorNoError_c	The new configuration is applied.

Usage

Call this function when SMAC is idle. Set bUseAutoMode to TRUE if PHY should switch between PAN's. Set bModifyDwell if PHY should also change the dwell time.

```
MLMEConfigureDualPanSettings(TRUE, TRUE, 0 , 3); //enable auto-mode and set dwell time to 0.5
* (3+1) = 2 ms
```

Implementation

This function validates the user request, fills and sends a message to the Multi PAN Manager (MPM) with the necessary updates.