MKW01 Simple Media Access Controller

Reference Manual

Document Number: MKW01SMACRM
Rev. 1
11/2015

How to Reach Us:

Home Page:
freescale.com

E-mail:
support@freescale.com

Chapter 1.
MKW01 SMAC introduction

Chapter 2.
Software architecture

Chapter 3.
Primitives

About This Book

This manual provides a detailed description of the Freescale MKW01 Simple Media Access Controller (MKW01 SMAC). This software is designed for use specifically with the MKW01 platform. The MKW01 is a highly-integrated, cost-effective, system-in-package (SiP), sub-1 GHz wireless node solution with FSK, GFSK, MSK, or OOK modulation-capable transceiver, and low-power ARM® Cortex® M0+ 32-bit MCU. The highly integrated RF transceiver operates over a wide frequency range including 315 MHz, 433 MHz, 470 MHz, 868 MHz, 915 MHz, 928 MHz, and 955 MHz, in the license-free Industrial, Scientific, and Medical (ISM) frequency bands.

The MKW01 SMAC software is predefined to operate in the 470 – 510 MHz , 863 – 870 MHz, 902 – 928 MHz, and 920 – 928 MHz bands.

Audience

This document is intended for application developers working on custom wireless applications based on the MKW01. The latest version of the Freescale MKW01 SMAC is available on freescale.com.

Organization

This document is organized into three chapters.

- Chapter 1, “MKW01 SMAC introduction” – this chapter introduces the MKW01 SMAC features and functionality.
- Chapter 2, “Software architecture” – this chapter describes the MKW01 SMAC software architecture.
- Chapter 3, “Primitives” – this chapter provides a detailed description of the MKW01 SMAC primitives.

Revision history

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

Revision history

Rev. number	Date	Substantive changes
0	03/2015	Initial release.
1	11/2015	FRDM-KW01, USB-KW01 mentions. Updated the SMAC Primitives.

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.

MKW01	MKW01 platforms (platforms hosting MKW01 MCU: FRDM-KW01, MRB-KW01, USB-KW01)
GUI	Graphical User Interface
MCU	Microcontroller
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
AES	Advanced Encryption Standard

References

The following sources were referenced to produce this book:

1. *MKW01Z128 Reference Manual* (document MKW01xxRM)

Chapter 1 MKW01 SMAC introduction

The Freescale MKW01 Simple Media Access Controller (MKW01 SMAC) is a simple ANSI C-based codebase, available as sample source code. The MKW01 SMAC is used for developing proprietary RF transceiver applications using Freescale's MKW01 sub-1 GHz transceiver with included MCU.
The MKW01 is a system-in-package (SIP) device, which includes an ARM® Cortex®-M0+ based MCU and a sub-1 GHz ISM band radio frontend device in an LGA-56 package. Features of the MKW01 include:

- MCU based on the 32-bit ARM Cortex-M0+ CPU with a full set of peripheral functions
- MCU that has 128 KB of flash and 16 KB of SRAM
- Fully featured, programmable sub-1 GHz transceiver, which supports the FSK, GFSK, MSK, GMSK, and OOK modulations schemes
- The MKW01 has internal and external connections between the MCU and the transceiver:
  - The MCU communicates with the transceiver through an internally connected SPI port.
  - Several transceiver status bits are also internally or externally connected to the MCU GPIO, and are capable of generating interrupt requests.

NOTE

It is highly recommended for you to be familiar with the MKW01 device. You can find additional details in the *MKW01Z128 Data Sheet* (MKW01Z128) and the *MKW01Z128 Reference Manual* (MKW01xxRM).

The MKW01 SMAC is a small codebase, which provides simple communication and test applications based on drivers, (802.15.4 compliant) PHY, and framework utilities, available as source code.
This environment is useful for hardware and RF debugging, hardware standards certification, and development of proprietary applications. The MKW01 SMAC is provided as a part of the example application demos available for the MKW01, and also as a standalone set of files.

To use any of the existing applications available in the MKW01 SMAC, download and open the available demonstration applications in the corresponding development environment (IDE).

The SMAC features include:

- Compact footprint:
  - Between 2 to 3 KB of flash required, depending on the configuration used.
  - Less than 500 B of RAM, depending on the configuration used.
- Low-power, proprietary, bidirectional RF communication link.
- The MKW01 radio enables checking of the preamble and the synchronization word, which reduces software overhead and memory footprint.
- Broadcast communication.
- Unicast communication – the MKW01 SMAC includes a node address 16-bit field. This enables the SMAC to perform unicast transmissions. To change the address of a node, modify the constant `gNodeAddress_c` inside the `SMAC_Interface.h` file, or call `SMACSetShortSrcAddress (uint16_t nwShortAddress)`. The address is set to 0xBEAD by default. Some of the demonstration applications enable you to change this address at runtime.

- Change of the current PAN. The SMAC packet uses a short 802.15.4-compliant header with a hardcoded configuration for frame control, which enables you to switch between PANs. The PAN address has 16 bits (`gDefaultPanID_c`). Modify this address by changing the default value in `SMAC_Interface.h` file, or by calling `SMACSetPanID` (`uint16_t nwShortPanID`).
- No blocking functions within the MKW01 SMAC.
- Enough flexibility to configure the packet header (preamble size, synchronization word size, and synchronization word value).
- Predefined settings at four different bands to initialize the SMAC protocol. The supported operating frequency bands are:
  - 863 – 870 MHz (Europe)
  - 902 – 928 MHz (US)
  - 920 – 928 MHz (Japan)
  - 470 – 510 MHz (China)
- Easy-to-use sample applications included.
- Light-weight, custom LBT algorithm.
- Light-weight, custom AA mechanism, which is transparent to you after enabling the feature.
- Encryption using the AES in cipher block chaining mode, with configurable initial vector and key.
- Configurable number of retries and a backoff interval.
- Inter-layer communication using SAPs.
- The MKW01 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 MKW01 SMAC-based demonstration applications

This is a list of the MKW01 SMAC-based demonstration applications:

- PC-based Connectivity Test application, which requires TERM. This application enables you to perform basic communication tests and several advanced XCVR tests.
- PC-based Wireless Messenger application, which requires TERM, and is presented in the form of a messenger-like application. This demonstration application highlights the Listen Before Talk and Automatic ACK mechanisms, providing the possibility to enable, disable, and configure them at runtime.
- PC-based Wireless UART application, which requires either 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) MKW01 platforms. It can be configured to use the previously mentioned mechanisms, but you must configure it at compile time.
- PC-based Low Power Demo application, which requires TERM. This application helps you to enable low-power modes on the MKW01 platforms.

1.2 Platform requirements

You can use the SMAC with any target application or board. However, Freescale provides several development platform designs, such as freedom development platform, modular reference board and USB dongle, with LEDs, push-buttons, and other modules included.

1.3 MCU resources used by the SMAC

The MKW01 contains an MCU and a transceiver in a single package. The SMAC does not use the 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, initialize the system protocol to configure the transceiver with correct functional settings, and set the SMAC state machine to known states. To initialize the SMAC, perform the following tasks:

1. Initialize the MCU interrupts and peripherals. This initialization is included in each demonstration application in the `hardware_init(void)` function, available as source code.
   — Initialize the LED, Keyboard, Serial Manager, Timers Manager, Memory Manager, and drivers, depending on the application needs.
   ```
   MEM_Init();
   TMR_Init();
   LED_Init();
   SerialManager_Init();
   ```
   — Initalize the PHY layer.
   ```
   Phy_Init();
   ```
2. Initialize the SMAC, set the SMAC state machine to default, configure addressing with default values, and initialize the RNG used for the first sequence number value and the random backoff.
   ```
   InitSmac();
   ```
3. Set the SAP handlers, so that the SMAC can notify the application on asynchronous events, on both the 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 `gMaxSmacSDULength_c` + `sizeof(packet type)`:
   ```
   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 MKW01 SMAC software architecture. The entire SMAC source code is always included in the application. The SMAC is primarily a set of utility functions or building blocks that you can use to build simple communication applications.

2.1 Block diagram

Figure 2-1 shows a simplified MKW01 SMAC-based stack block diagram.

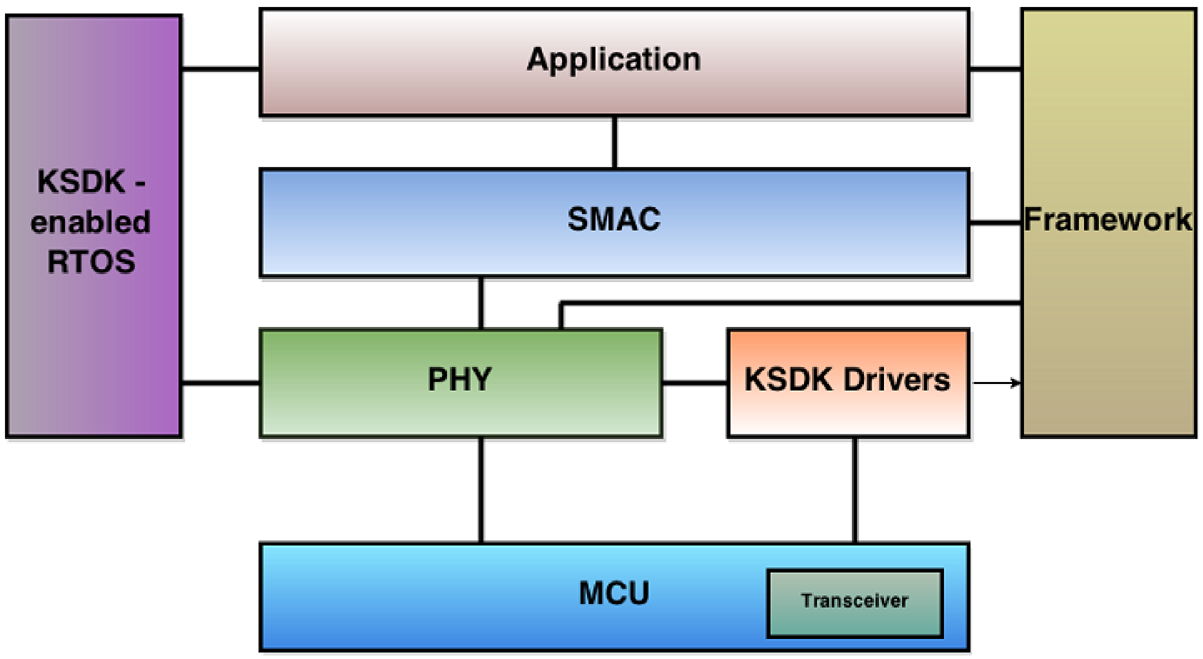

Figure 2-1. SMAC system decomposition

The application programming interface (API) is implemented in the MKW01 SMAC as a C header file (`.h`) that enables access to the code. The code includes the API for 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

The MKW01 SMAC projects support only the MKW01-based target boards designated in the files.

2.2 MKW01 SMAC data types and structures

The MKW01 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 MKW01 SMAC:

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 MKW01 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 the MKW01 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	Maximum number of bytes to be received.
rxStatus	Indicates the reception state. See `rxStatus_t` data type for more details.
u8DataLength	Number of received bytes.
smacPdu	The MKW01 SMAC protocol data unit.
smacHeader	The SMAC structure that defines the header used. Freescale recommends that you don’t modify this structure directly, but through the associated functions.

Usage

Application uses this data type in the following way:

1. Declare a buffer to store a packet to be received OTA. Freescale recommends that the size of this buffer is at least as long as the largest packet to be received by the application.
2. Declare a pointer of the `rxPacket_t` type.
3. Initialize the pointer to point to the buffer declared in the first step.
4. Initialize the `u8MaxDataLength` member of the packet structure. The SMAC filters 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 `smacErrors_t` type to store the result of executing the `MLMERXEnableRequest` function.

2.2.3 smacHeader_t

This structure defines the variable used for the MKW01 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 must 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 change it through the associated function.
destAddr	The short destination address.
srcAddr	The short source address.

Usage

Freescale recommends that you don’t 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 MKW01 SMAC does not use this.
rxProcessingReceptionStatus_c	This state is set when the MKW01 SMAC is in the middle of receiving a packet.
rxSuccessStatus_c	This is one of the possible finish conditions for a packet that is successfully received, and can be checked by the indication functions.
rxTimeOutStatus_c	This is another of the possible finish conditions for a timeout condition, and can be checked by the indication functions.
rxAbortedStatus_c	This status is set when the SMAC drops a packet (on SMAC-specific criteria) validated by the PHY, and the enter reception request is performed with a timeout.
rxMaxStatus_c	This element indicates the total number of possible reception states.

2.2.5 smacPdu_t

This type defines the SMAC basic protocol data unit:

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

Members

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

2.2.6 txPacket_t

This structure defines the type of variable to be transmitted by the MKW01 SMAC. It is located in the `SMAC_Interface.h` file, and it 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	The SMAC structure that defines the header used. Freescale recommends that you don’t modify this structure directly, but through the associated functions.
smacPdu	The MKW01 SMAC protocol data unit.

Usage

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

1. Declare a buffer to store the packet to be transmitted OTA. Freescale recommends that the size of this buffer is at least as long as the largest packet to be transmitted by the application.
2. Declare a pointer of the `txPacket_t` type.
3. Initialize the pointer to point to the buffer declared in the first step.
4. Copy the desired data into the payload.
5. Set `u8DataLength` to the size of the payload (in bytes).
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;
SMACFillHeader(&(TxPacket->smacHeader), 0xFFFF);
FLib_MemCpy(TxPacket->smacPdu.smacPdu, dataToBeSentBuffer, payloadSizeBytes);
TxPacket->u8DataLength = payloadSizeBytes;
DataRequestResult = MCPSDataRequest(TxPacket);
```

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

2.2.7 channels_t

Definition of the RF channels. The number of a channel varies in each defined operating band for the sub-1 GHz stacks, and it is fixed for the 2.4 GHz. The first logical channel in all bands is 0 for the sub-1 GHz, and 11 for the 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 MKW01 SMAC API functions, and is located in the `SMAC_Interface.h` file. Some of the messages sent by the 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	The MKW01 SMAC accepts the request and processes it. This return value does not necessarily mean that the action requested was successfully executed. It only means that it was accepted for processing by the MKW01 SMAC. This value is also used as a return status in the SMAC to application SAPs. For example, if a packet was succesfully sent, the message has a data-confirm field with this status. This value is also returned in the CCA confirm message when the scanned channel is found idle.
gErrorBusy_c	This constant is returned when the MKW01 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. The CCA confirm message can also have this value when 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 MKW01 SMAC operations when the MKW01 SMAC is not initialized, or requesting to disable the RX when the SMAC is 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 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 for enabling / disabling the LBT mechanism.
autoAck	`bool_t` value for enabling / disabling the AA mechanism.
retryCountCCAFail	This value specifies the number of times the MKW01 SMAC attempts to retransmit a packet when the LBT is enabled and channel is found busy.
retryCountAckFail	This value specifies the number of times the MKW01 SMAC attempts to retransmit a packet if the AA is enabled and no acknowledgement message is received in the expected time frame.

2.2.10 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 made at the application level, and this enumeration is used only as a reference for designing the test modes.

2.2.11 packetConfig_t

```
typedef struct packetConfig_tag
{
  uint16_t u16PreambleSize;
  uint8_t  u8SyncWordSize;
  uint8_t* pu8SyncWord;

} packetConfig_t;
```

Members

u16PreambleSize	The preamble size.
u8syncWordSize	The size of the synchronization word.
pu8SyncWord	Pointer to an array containing the value of the synchronization word.

Usage

Declare a variable of the `packetConfig_t` type. Fill in the required information, and call `MLMEPacketConfig` with a pointer to the structure. Optionally, you can store the return value in the `smacErrors_t` variable.

See the reference manual for valid lengths of the preamble and synchronization word.

2.2.12 smacRFModes_t

```
typedef enum smacRFModes_tag
{
  gRFMode1_c = gPhyMode1_c,
  gRFMode2_c = gPhyMode2_c,
  gRFMode3_c = gPhyMode3_c,
  gRFMode4_c = gPhyMode4_c,
  gRFMode5_c = gPhyMode1ARIB_c, /*ARIB mode 1*/
  gRFMode6_c = gPhyMode2ARIB_c, /*ARIB mode 2*/
  gRFMaxMode_c
} smacRFModes_t;
```

Members

gRFModeX_c	The X ranging from 1 to 4 corresponds to PHY modes 1 to 4. You must ensure that the frequency band used supports the mode to be used.
gRFMode5_c	This corresponds to PHY mode 1, in case you want to use the Japan frequency band with ARIB mode 1.
gRFMode6_c	This corresponds to PHY mode 2, in case you want to use the Japan frequency band with ARIB mode 2.

Usage

Call `MLMESetPhyMode` with the desired enumeration member. Optionally, you can store the return value in the `smacErrors_t` variable.

2.2.13 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 the SMAC. Call `SMAC_SetIVKey` with two 16-byte buffer pointers as parameters to change the SMAC initial vector and encryption key settings.

2.3 MKW01 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, and so on).
- 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. 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 the *smacErrors_t* element. See Section 2.2.8, “smacErrors_t.”
2	`gMcpsDataInd_c`	smacDataInd_t	*u8LastRxRssi* • Value indicating the RSSI obtained during the reception. *pRxPacket* • Pointer to the packet passed as a parameter to *MLMERXEnableRequest.*
3	`gMlmeCcaCnf_c`	smacCcaCnf_t	Contains a *smacErrors_t* element. See Section 2.2.8, “smacErrors_t.”

Table 2-1. Message types and associated data structures (continued)

Index	Message type	Associated data structures	Description
4	`gMlmeEdCnf_c`	smacEdCnf_t	*status* • This is a *smacErrors_t* element. If the PHY succesfully performs the ED, its 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	—

Considering all of the above, 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 the application specifies the functions to handle asynchronous responses, the SAP handlers aquire the value of those functions. Here are the definitions of the handlers:

```
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

This chapter provides a detailed description of the MKW01 SMAC primitives associated with the MKW01 SMAC application API.

3.1 MCPSDataRequest

This data primitive is used for sending an OTA packet. This is an asynchronous function, which means that it asks the MKW01 SMAC to transmit an OTA packet, but the transmission can continue even 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 is performed.
gErrorOutOfRange_c	One of the members in the `pTxMessage` structure is out of range (invalid buffer size or data buffer pointer being NULL).
gErrorBusy_c	The radio is performing another action, and it cannot attend this request.
gErrorNoValidCondition_c	The MKW01 SMAC was not initialized.
gErrorNoResourcesAvailable_c	The PHY cannot process the MKW01 SMAC request, so the MKW01 SMAC cannot process it either, or the memory manager is unable to allocate another buffer.

Usage

- Initialize the SMAC before calling this function.
- Declare a variable of the `smacErrors_t` type 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 result of calling this function is different than `gErrorNoError_c`, the application handles the error returned. For instance, if the result is `gErrorBusy_c`, the application waits for the radio to finish the 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

The `MCPSDataRequest` primitive creates a message for the PHY task, and fills it according to your configurations made prior to this call, and according 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 the current operation is TX. It does not explicitly check whether the 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 you want the application to switch from TX into idle.

Prototype

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

Arguments

None.

Returns

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

Usage

Call `MLMETXDisableRequest()`.

Implementation

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

3.3 MLMEConfigureTxContext

This function helps you to enable / disable the LBT and AA mechanisms, and to configure the number of retries in case the 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 succesfully applied.
gErrorNoValidCondition_c:	The number of retries is set, but the corresponding mechanism boolean is set to FALSE.
gErrorOutOfRange_c:	The number of retries exceeds `gMaxRetriesAllowed_c`.

Usage

- Declare a structure of the `txContextConfig_t` type.
- Set the desired values of the members.
- Call `MLMEConfigureTxContext` with the address of the declared structure as parameter.
- Capture the return value in the `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 in which the SMAC will handle data requests and responses from PHY, according to the parameters described by the `txContextConfig_t` structure. The requests forwarded by the SMAC to the PHY depend on addressing and `txContextConfig_t` information.

3.4 MLMERXEnableRequest

This function places the radio into the 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 are stored.
uint32_t u32Timeout:	A 32-bit timeout value in symbol duration. Symbol duration depends on the bitrate (on the PHY mode).

Returns

gErrorNoError_c	Everything is OK, and the reception is performed.
gErrorOutOfRange_c	One of the members in the `rxPacket_t` structure is out of range (invalid buffer size or data buffer pointer being NULL).
gErrorBusy_c	The radio is performing another action and cannot attend this request.
gErrorNoValidCondition_c	The MKW01 SMAC is not initialized.
gErrorNoResourcesAvailable_c	The PHY cannot process the MKW01 SMAC request, so the MKW01 SMAC cannot process it neither.

Usage

- Initialize the SMAC before calling this function.
- Declare a variable of the `smacErrors_t` type 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 the `MLMERXEnableRequest` function.
- If the result of calling the function is different to `gErrorNoError_c`, the application handles the error returned. For instance, if the result is `gErrorBusy_c`, the application waits for the radio to finish the 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` means that the receiver did not go into the receive mode.
- A 32-bit timeout value of 0 causes the receiver to never timeout, and to stay in the receive mode until a valid data packet is received or the `MLMERXDisableRequest` function is called.
- To turn the receiver off before a valid packet is received, you can call the `MLMERXDisableRequest` function.
- If the timeout is not zero, and a valid packet longer than `u8MaxDataLength` is received, the SMAC sends a data indication message, and sets the `rxAbortedStatus_c` in the `rxStatus_t` field of the `rxPacket_t` variable.
- When using security, the maximum payload allowed for transmission is `gMaxSmacSDULength_c`; for reception, you must configure the `u8MaxDataLength` field to `gMaxSmacSDULength_c` + 16 (the maximum number of padding bytes for the encryption algorithm), so that the SMAC will not filter out the received packets of the `gMaxSmacSDULength_c` size.

Implementation

This primitive creates a message for the 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 the PHY to enable the *gPhyPibRxOnWhenIdle* attribute.

3.5 MLMERXDisableRequest

Returns the radio from the receive mode to the idle mode.

Prototype

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

Arguments

None

Returns

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

Usage

Call `MLMERXDisableRequest()`.

NOTE

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

Implementation

This function creates a message for the PHY, and, if the timeout value from `MLMERXEnableRequest` is 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 currently requested action, puts the PHY to the idle state, and sets the transceiver to the standby mode. It also disables any previous timeout programmed.

3.6 MLMELinkQuality

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

Prototype

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

Arguments

None.

Returns

uint8_t	An 8-bit value, which represents the link quality value. Returns the result in `smacLastDataRxParams.linkQuality`.
Zero	The MKW01 SMAC is not initialized.

Usage

Call `MLMELinkQuality()`.

Implementation

This function reads the value stored 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 function sets the frequency, on which the radio transmits or receives.

Prototype

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

Arguments

channels_t newChannel:	An 8-bit value, which represents the requested channel.

Returns

gErrorNoError_c	The channel setting was performed.
gErrorBusy_c	The MKW01 SMAC is busy with other radio activity, such as transmitting / receiving data, or performing a channel scan.
gErrorOutOfRange_c	The requested channel is invalid.
gErrorNoValidCondition_c	The MKW01 SMAC is not initialized.

Usage

Call `MLMESetChannelRequest(newChannel)`.

NOTE

Make sure to enter a valid channel between 0 and (`gTotalChannels` – 1).

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	An 8-bit value of the desired output power. Values in the range of 0 – 31 are required.

Returns

gErrorOutOfRange_c	The `u8Power` exceeds the maximum power value of `gMaxOutputPower_c` (0x1F).
gErrorBusy_c	The MKW01 SMAC is busy, or the PHY is busy.
gErrorNoError_c	The action is performed.
gErrorNoValidCondition_c	The MKW01 SMAC is not initialized.

Usage

Call `MLMEPAOutputAdjust(u8PaValue)`.

NOTE

Be sure to enter a valid value for the PA output adjustment.

3.10 MLMEPhySoftReset

The `MLMEPhySoftReset` function is called to perform a software reset of the PHY and MKW01 SMAC state machines.

Prototype

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

Arguments

None.

Returns

gErrorNoError_c	The action is performed.
gErrorNoValidCondition_c	The MKW01 SMAC is not initialized.

Usage

Call `MLMEPHYSoftReset()`.

Implementation

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

3.11 MLMEScanRequest

This function creates an ED request message for the PHY. If the channel passed as a 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 OK, and the scan is performed.
gErrorBusy_c	The radio is performing another action.
gErrorNoValidCondition_c	The MKW01 SMAC is not initialized.

Usage

Call the function with the selected channel to be scanned:

```
MLMEScanRequest(u8ChannelToScan);
```

NOTE

Make sure to enter a valid channel. Make sure to 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. The CCA is performed on the active channel (set with `MLMESetChannelRequest`). The result is received in a message passed through the SMAC to the application management SAP.

Arguments

None.

Returns

gErrorNoValidCondition_c	The MKW01 SMAC is not initialized.
gErrorBusy_c	Either the SMAC or the PHY is busy and cannot process the request.
gErrorNoError_c	Everything is OK, and the request is processed.

Usage

Call the function. The application can store the return value in the `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 the SMAC becomes idle.

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

Implementation

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

3.13 MLMESetPreambleLength

This function updates the number of the preamble repetitions.

Arguments

uint16_t u16preambleLength	The new value of the number of preamble repetitions.

Returns

gErrorNoValidCondition_c	The SMAC is not initialized.
gErrorBusy_c	The SMAC is busy and cannot process the request.
gErrorNoError_c	The request is processed.

Usage

Call the function. The application can store the return value in the `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 the SMAC becomes idle.

3.14 MLMESetSyncWordSize

This function updates the size of the synchronization word (maximum size is eight bytes).

Arguments

uint8_t u8syncWordSize	The size of the synchronization word in bytes.

Returns

gErrorNoValidCondition_c	The SMAC is not initialized.
gErrorBusy_c	The SMAC is busy, and cannot process the request.
gErrorOutOfRange_c	The requested size is out of the valid range.

gErrorNoError_c	The request is processed.

Usage

Call the function. The application can store the return value in the `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 the SMAC becomes idle.

3.15 MLMESetSyncWordValue

This function updates the value of the synchronization word.

Arguments

uint8_t* u8SyncWordValue	Pointer to the buffer containing the new values for the synchronization word.

Returns

gErrorNoValidCondition_c	The SMAC is not initialized.
gErrorBusy_c	The SMAC is busy, and cannot process the request.
gErrorNoError_c	The request is processed.

Usage

Call the function. The application can store the return value in the `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 the SMAC becomes idle.

3.16 MLMEPacketConfig

This function calls the three above functions with the parameters configured in the `packetConfig_t` structure, passed by the address as parameter.

Arguments

packetConfig_t* pPacketCfg	Pointer to the `packetConfig_t` structure containing the new values for preamble length, synchronization, word size, and values.

Returns

gErrorNoValidCondition_c	The SMAC is not initialized.
gErrorBusy_c	The SMAC is busy, and cannot process the request.
gErrorOutOfRange_c	One of the configuration parameters is out of range.
gErrorNoError_c	The request is processed.

Usage

Call the function. The application can store the return value in the `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 the SMAC becomes idle.

3.17 MLMESetAdditionalRFOffset

This function sets the frequency drift in a number of Fsteps (57 Hz on the 30 MHz platforms, 61 Hz on the 32 MHz platforms) passed as parameters, to fine-tune the central frequency of the channel on the MKW01 platforms. The frequency is updated with the next `MLMESetChannelRequest` call.

Arguments

uint32_t additionalRFOffset	A signed value of the drift, converted to `uint32_t`.

Returns

gErrorNoValidCondition_c	The SMAC is not initialized.
gErrorNoResourcesAvailable_c	The calibration feature is not enabled.
gErrorNoError_c	Everything went fine.

Usage

Use the Connectivity Test (continuous, unmodulated TX) and a spectrum analyzer to determine the real channel frequency. Compute the drift, and divide it to Fsteps. Call this function with the result.
The application can store the return value in the `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 the SMAC becomes idle.

3.18 MLMEGetAdditionalRFOffset

This function returns the last stored frequency drift, or 0, if the feature is not enabled.

Arguments

None.

Returns

uint32_t	The signed value of the offset, converted to `uint32_t`.

3.19 MLMESetAdditionalEDOffset

This function sets the ED compensation value in case the calibration feature is enabled.

Arguments

uint8_t additionalEDOffset	A signed value of the ED compensation, converted to `uint8_t`.

Returns

gErrorNoValidCondition_c	The SMAC is not initialized.
gErrorNoResourcesAvailable_c	The calibration feature is not enabled.
gErrorNoError_c	Everything went fine.

Usage

Use the Connectivity Test (Calibrate ED Measurement) and a signal generator to send a signal with known power. Input the value of the known output power in dBm and start the test to determine the measured ED value. The test automatically calls this function to notify PHY about the compensation. The application can store the return value in the `smacErrors_t` variable, and handle the error in case it occurs.

3.20 MLMEGetAdditionalEDOffset

This function returns the last stored energy detection compensation, or 0, if the feature is not enabled.

Arguments

None.

Returns

uint8_t	The signed value of the compensation, converted to `uint8_t`.

3.21 SMACSetShortSrcAddress

This function creates a message of a set PIB request type, requesting the PHY to change the short source address of the node. If the message is succesfully passed to the PHY, the SMAC will set its own source address variable to the new value, so that when the `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 cannot handle this request.
gErrorBusy_c	The PHY is busy, and cannot process the request.
gErrorNoError_c	Everything is OK, and the request is processed.

Usage

Call the function with the desired address. The application can store the return value in the `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 the PHY becomes idle.

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

Implementation

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

3.22 SMACSetPanID

This function creates a message of a set PIB request type, requesting the PHY to change the short PAN address of the node. If the message is succesfully passed to the PHY, the SMAC will set its own PAN address variable to the new value, so that when the `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 cannot handle this request.
gErrorBusy_c	The PHY is busy, and cannot process the request.

gErrorNoError_c	Everything is OK, and the request is processed.

Usage

Call the function with the desired address. The application can store the return value in the `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 the PHY becomes idle.

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

Implementation

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

3.23 SMACFillHeader

This function has no interaction with the PHY layer. Its purpose is to help the application in configuring the addressing for a packet to be sent. It fills the packet header with the updated addressing and hardcoded configuration values, and adds the destination address passed as a parameter.

Arguments

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

Returns

None.

Usage

Call the function when it is the first time the application uses the `txPacket_t` variable, or when 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 the default, hardcoded frame control, and sequence number values. It also adds the addressing information (configured by calling `MLMESetShortSrcAddress` and `MLMESetPanID`), and the destination address passed as a parameter.

3.24 SMAC_SetIVKey

This function sets the initial vector and encryption key for the encryption process, when `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 a size of 16 B. Fill one of them with key information, and the other one with initial vector information. Call this function with pointers to the buffers as parameters.

3.25 Smac_RegisterSapHandlers

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

Arguments

SMAC_APP_MCPS_SapHandler_t	pSMAC_APP_MCPS_SapHandler: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 the value for this parameter, because this version of SMAC does not support multiple instances.

Returns

None.

Usage

Implement two functions that meet the constrains 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 internal SMAC function handlers with the ones registered by the application. Whenever an asynchronous response must be passed from the SMAC to the application, call the internal handlers, which in turn call the ones defined by the application.