www.cohdawireless.com

Cohda*Mobility* MKx Radio
LLCremote API Specification

Date:	9 Feb, 2018
Version:	1.3
Reference:	CWD-MKx-0208
Prepared for:	
Prepared by:	Jason Boyd Wade Farrell
Authorised By:	Paul Gray Paul Alexander Fabien Cure
Distribution:	External

ABN 84 107 936 309

COMMERCIAL-IN-CONFIDENCE

Cohda Wireless Pty Ltd
82-84 Melbourne St,
North Adelaide, SA 5006
Australia

P +61 8 8364 4719
F +61 8 8364 4597

www.cohdawireless.com

Change Log

Version	Date	Comments
0.1	06/03/2013	Initial version for internal review
0.2	08/03/2013	Internal review updates
0.3	13/06/2013	Expanded introduction
0.4	07/11/2014	Security variant added. Updates made to data structures to match MKxAPI.h
0.5	16/09/2015	Updated LLCremote API section and added MKx_Recv to API section
0.6	25/11/2015	Updated LLCremote API section to match changes made in Release 13
0.7	02/12/2015	Clarify recommended value for '<headroom>'
0.8	21/12/2015	Updated LLCremote API section to match changes made in Release 13.1
0.9	17/06/2016	Updated transmit event structures and address matching control for Release 13.2
1.0	23/06/2016	Pre-release, updated GetTSF and SetTSF calls and data structures in order to support timing advertisements. TxFlush also updated.
1.1	14/07/2016	Added MediumBusyTime and updated NumShortRetries and NumLongRetries types
1.2	08/02/2017	Updated MKx_TxReq description to mention what MAC header updates are performed. Added RSSI calibration function.
1.3	09/02/2018	Added new security features (decompress key, reconstruct key). Added support for up to two MKx radio devices connected to the LLC.

Table of Contents

1 Introduction

This document provides the API Specification for the **Cohda***Mobility* MKx radios. The document firstly outlines the architecture of the LLC system which incorporates the following components:

- **Cohda***Mobility* MKx PHY,
- **Cohda***Mobility* Multi-Context MAC,
- LLC layer.

This is followed by the API for the LLC layer. Note the LLC layer supports up to two **Cohda***Mobility* MKx radio devices connected to the host.

The LLC is built upon the functionality of the **Cohda***Mobility* Multi-Context MAC layer, which provides both the dual-radio ETSI TC-ITS MAC and the time-synchronised context-switching required by the channel-switching IEEE1609.4 MAC. The LLC is designed to be integrated with (and controlled by) Cohda's V2Xnet network layer software or any 3rd party network stack implementation.

2 MAC Hardware/Software Architecture

The top-level architecture of the MKx design is shown in Figure 1. This diagram shows the hardware and software components of the MKx architecture, and how they relate to one another.

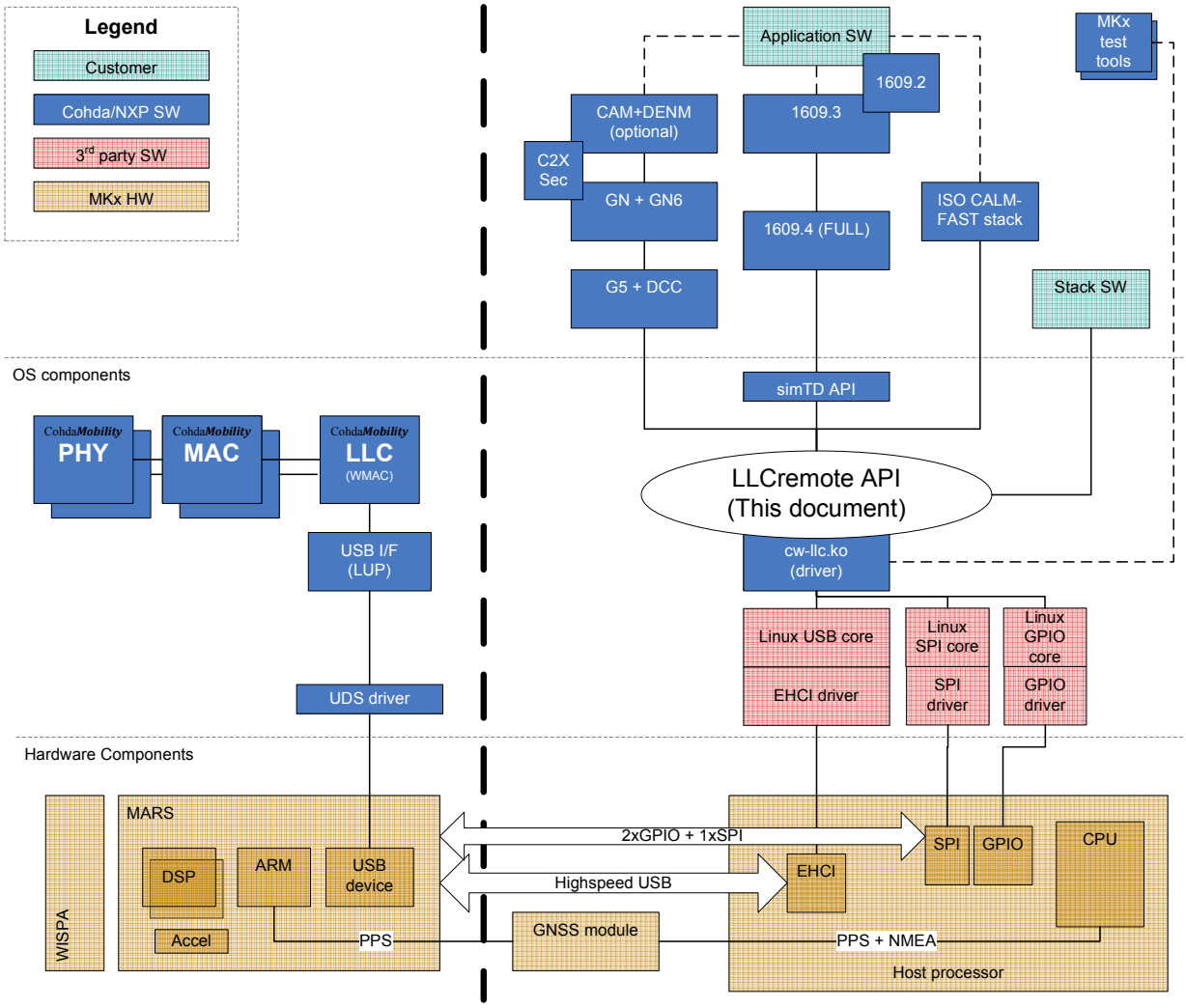


Figure 1: MKx architecture

In an ETSI TC-ITS-style dual-radio implementation two independent **Cohda***Mobility* MAC & PHYs are available (and the UTC timing function is not strictly required), while in an IEEE 1609.4-style single radio implementation only one PHY may be used (and the UTC timing function is required to coordinate channel switches). In the dual radio configuration it is possible to simultaneously receive on both interfaces (although not while one of the interfaces is transmitting), and simultaneously transmit on both interfaces (subject to antenna selection).

The LLC is built upon the facilities provided by the Multi-Context MAC, which provides time-synchronised mode-switching MAC functionality. The functionality of the above functional units is outlined in the following sub-sections.

2.1 LLCremote

The LLCremote layer is implemented as a Linux kernel module that runs on the host processor, running in kernel-space.

This layer provides the following functionality.

- 802.11 firmware download to the SAF5x00
  - Interfacing to the operating system SPI core API
  - Interfacing to the operating system GPIO core API
- USB interfacing to the SAF5x00
  - Interfacing to the operating system USB core API
  - Interfacing to the operating system networking API
- LLCremote functional API (detailed in section 3)
  - Implements the interfacing to multi-context MAC functionality, including the MLME & PLME interface to the underlying MAC modules

Also provided is a user-space library (libLLC.so) which provides user-space access to the LLCremote API running in kernel-space. This has some additional API features documented in section 3.4.2 which are specific to the user-space library.

2.2 Multi-Context MAC (LLC)

The Multi-Context MAC provides the abstraction upon which the ETSI TC-ITS dual-radio or IEEE 1609.4 MAC layer is implemented. It provides the following functionality.

- Registration of multiple MAC/PHY operating contexts.
  - A MAC/PHY mode context (MPMC) specifies the MAC and PHY operating parameters/configuration while operating in that context, e.g. channel frequency, PHY bandwidth mode, EDCA parameters, receive packet handler functions.
- Configuration of context switching behaviour
  - E.g., periodic switching between channel configurations, synchronised with the MAC TSF, including guard intervals.
- Per-channel configuration statistics & measurements
  - Periodic statistic generation (synchronized with context switches).
  - MAC internal data (e.g. QoS duplicate detection, current transmit MCS, power level and antenna(s), RSSI, time of last rx/tx packet).
  - Channel measurements (Channel busy ratio, Average Idle power)
- UTC Timing Function
  - Measures and maintains TSF timing synchronisation with an external GPS PPS signal and UTC timestamps provided by the LLC Remote
- MAC Header Manipulations
  - Duration field is overwritten (for data frames only)
  - Sequence number is incremented and overwritten (note MAC maintains sequence numbers for each individual QoS queue. This is performed for all frame types except for control frame types).
  - QoS Control ACK policy overwrite if multi-cast frame selected

3 LLCremote API

3.1 Asynchronous function call interface

The interface to the LLC is implemented via an asynchronous function call interface. 'Asynchronous' means that all function calls are non-blocking. In cases where the action can be performed immediately and atomically, the confirmation is reported in the request function's return code. Alternatively, the initiation of an action is performed by a call to a 'request' function and the result of the action is reported by a corresponding 'confirm' callback.

Some of the API actions exist as request/confirm pairs and can be identified by their naming: A function MKx_<action>Req() corresponds with the MKx_<action>Cnf() and MKx_<action>Ind() callbacks.

All the LLC callbacks, are invoked in a non-interrupt context to allow the stack a modicum of leeway inside the callback processing (can use mutexes etc.), but should not perform long complex operations as this will block all further LLC processing.

3.2 MKx handles

The handle provided by the LLC consists of the following elements:

- State
- Config
- API request and indication (callback) function pointers

```
/// MKx handle
typedef struct MKx
{
  /// 'Magic' value used as an indicator that the handle is valid
  uint32_t Magic;
  /// Private pointer for client
  void *pPriv;
  /// Stats counters, etc. (read only)
  struct MKxState State;
  /// Currently active configuration (read only)
  struct MKxConfig Config;
  /// Function pointers for requests and callback indications
  struct API API;
} tMKx;
```

A handle is created and destroyed with the MKx_Init () and MKx_Exit () functions. Once created, the stack can modify the handle's configuration using the MKx_Config() function. The LLC will verify the provided configuration before committing it to the hardware. The new active configuration is then copied into the 'Config' portion of the handle's structure. If two MKx radio devices are connected to the host, up to two separate handles can be created.

The general lifetime of a handle is the following:

1. The stack invokes the MKx_Init() function to create a handle.
    1. The underlying MAC resources are initialised and reserved. The handle is created by the LLC and returned to the stack

2. The stack copies the 'Config' field structure within the handle (to get the defaults) and updates the configuration before calling MKx_Config()
   - The LLC verifies the configuration and once applied, copies the new configuration into the ‘Config’ portion of the handle
3. The LLC updates the handle statistics during its processing
4. While a handle is enabled, the stack may only update the callback function pointers

For detailed concrete examples of handle usage see Section 4.

3.3 802.11p data format

The TxReq() and RxInd() functions are used to communicate 802.11 frame data between the LLC and upper-layer stack entities. The format of packets transported on this interface is shown below. All packets have either MKxTxPacket or MKxRxPacket headers, followed by the raw 802.11 frame data. Figure 2 shows the required layout within in a Linux sk_buff.

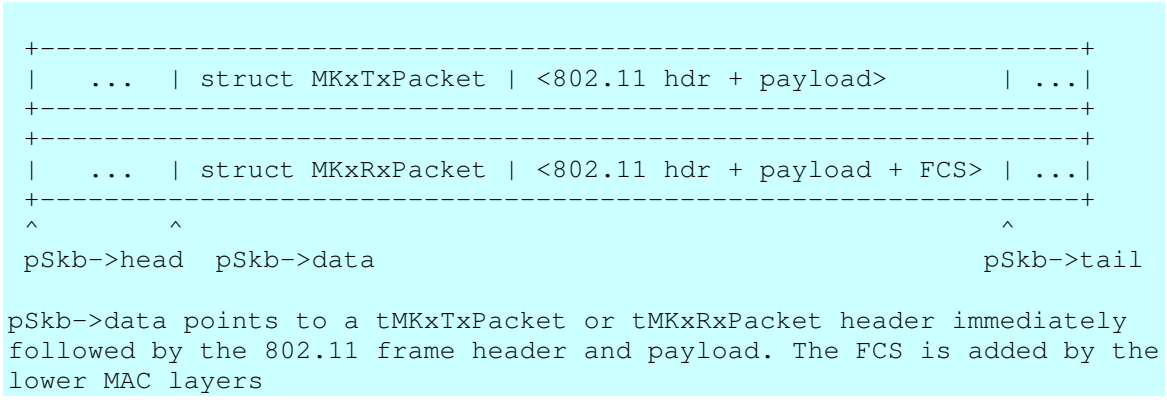


Figure 2: LLC 802.11 frame interface packet formats

For transmit packets, the LLC checks the MKxTxPacket and 802.11 headers for consistency before submitting them to the specific MAC queue so invalid or malformed packets are not transmitted.

3.4 API

3.4.1 General API Functions & Callbacks

The following functions and callbacks are not specific to either kernel space or user space usage; they are relevant for both.
The following sections outline the LLC API functions and callbacks. Note that the usage of ‘stack” below indicates the object using the API.

3.4.1.1 MKx_Init()

Initialize the firmware and LLC interface, and returns a handle to be used when calling MKx_* functions. Once the interface is no longer required, the handle may be closed via a call to MKx_Exit(). If two devices are connected to the host, it is possible to obtain a separate handle for each device, by setting the DevId accordingly.

```
tMKxStatus MKx_Init ( uint8_t DevId, tMKx ** ppMKx )
```

Parameters:

DevId Device number (0..1)
ppMKx **MKx** handle to initialize

Returns:

MKX_STATUS_SUCCESS (0) or a negative error code

This function will:

- Reset and download the SDR firmware if present
  - The SDR firmware image may be compiled into the driver as a binary object
- Initialize the USB or Ethernet interface

3.4.1.2 MKx_Exit()

Closes the communication path with the LLC, previously established via a call to MKx_Init(). After calling this function, the handle *pMKx must not be used.

```
tMKxStatus MKx_Exit ( tMKx * pMKx )
```

Parameters:

pMKx **MKx** handle

Returns:

MKX_STATUS_SUCCESS (0) or a negative error code

3.4.1.3 MKx_Config()

A function invoked by the stack to request a re-configuration of a particular MKx radio.

```
typedef tMKxStatus(* fMKx_Config)(struct MKx *pMKx,
                                  tMKxRadio Radio,
                                  tMKxRadioConfig *pConfig)
```

Parameters:

pMKx **MKx** handle
Radio the selected radio
pConfig Pointer to the new configuration to apply

Returns:

MKX_STATUS_SUCCESS if the request was accepted

```
// Get the current/default config
tMKxRadioConfig Cfg = {0,};
memcpy(&Cfg, &(pMKx->Config.Radio[MKX_RADIO_A]), sizeof(Cfg));
// Update the values that we want to change
Cfg.Mode = MKX_MODE_SWITCHED;
Cfg.Chan[MKX_CHANNEL_0].PHY.ChannelFreq = 5000 + (5 * 178);
Cfg.Chan[MKX_CHANNEL_1].PHY.ChannelFreq = 5000 + (5 * 182);
...
// Apply the configuration
Res = MKx_Config(pMKx, MKX_RADIO_A, &Cfg);
```

Note: The MKx_NotifInd() callback is invoked separately to notify the stack if the configuration was successfully applied or if there was an error.

3.4.1.4 MKx_TxReq()

A function invoked by the stack to request the transmission of an 802.11 frame.

```
typedef tMKxStatus(* fMKx_TxReq)(struct MKx *pMKx,
                                 tMKxTxPacket *pTxPkt,
                                 void *pPriv)
```

Parameters:

pMKx **MKx** handle
pTxPkt The packet pointer (including tx header)
pPriv Pointer to provide when invoking the **fMKx_TxCnf** callback

Returns:

MKX_STATUS_SUCCESS if the transmit request was accepted

Note:

The following MAC header parameters are manipulated by the MAC layer:
- Duration field is overwritten (for data frames only)
- Sequence number is incremented and overwritten (note MAC layer maintains sequence numbers for each individual QoS queue. This is performed for all frame types except for control frame types).

In addition, unicast frames are determined by the destination address top byte LSB being set to 0.
- For unicast QoS frames, the ACK policy is determined by the QoSControl setting.
- For unicast non-QoS frames, the ACK policy is always true.
- For multi-cast QoS frames, the ACK policy of the QoSControl field is cleared by the MAC layer to be always false

When applicable, RTS, CTS, and Ack control frames are created by the MAC layer and are not made available via the LLC interface.

The buffer must lie in DMA accessible memory and there is usually some relation between pTxPkt and pPriv. A possible stack implementation is shown below:

```
Len = <headroom> + sizeof(struct MKxTxPacket) + <802.11 Frame length>;
pSkb = alloc_skb(Len, GFP_DMA);
pTxPkt = pSkb->data;
pTxPkt->Length = <802.11 Frame length>
 /// +-------------------------------------------------------------------+
 /// |     ...   | struct MKxTxPacket | <802.11 hdr & payload> |  ...  |
 /// +-------------------------------------------------------------------+
 /// ^           ^                                                       ^
 /// pSkb->head    pSkb->data                                          pSkb->tail
 /// pSkb->data points to a tMKxTxPacket or tMKxRxPacket header immediately
 /// followed by the 802.11 frame header and payload.
 /// The FCS is added by the lower MAC layers
 ...
Res = MKx_TxReq(pMKx, pTxPkt, (void *)pSkb);
```

Note: The recommended value for '<headroom>' is in the provided codebase's #define LLC_DEV_HEADROOM

3.4.1.5 MKx_TxCnf()

A callback invoked by the LLC to notify the stack that the provided transmit packet was either successfully transmitted or failed to be queued/transmitted.

```
typedef tMKxStatus(* fMKx_TxCnf)(struct MKx *pMKx,
                                 tMKxTxPacket *pTxPkt,
                                 const tMKxTxEvent *pTxEvent,
                                 void *pPriv)
```

Parameters:

pMKx	**MKx** handle
pTxPkt	As provided in the **fMKx_TxReq** call
pTxEvent	A pointer to the event data generated for the TxReq
pPriv	As provided in the **fMKx_TxReq** call

Returns:

MKx_STATUS_SUCCESS if the 'confirm' was successfully handled. Other values are logged for debug purposes.

Note:

The pTxEvent should not be modified and will be freed after return from this callback.

Continuing the example from **fMKx_TxReq**...

```
{
  ...
  free_skb(pPriv);
  return MKX_STATUS_SUCCESS;
}
```

3.4.1.6 MKx_TxFlush()

A function invoked by the stack to request the cancellation of all pending transmit packets in the specified queue or multiple queues.

```
typedef tMKxStatus(* fMKx_TxFlush)(struct MKx *pMKx,
                                   tMKxRadio RadioID,
                                   tMKxChannel ChannelID,
                                   tMKxTxQConfig TxQueue)
```

Parameters:

pMKx	**MKx** handle
RadioID	The specific radio (MKX_RADIO_A or MKX_RADIO_B)
ChannelID	The specific channel (MKX_CHANNEL_0 or MKX_CHANNEL_1)
TxQueue	The specific queue (MKX_TXQ_COUNT for all)

Returns:

MKX_STATUS_SUCCESS if the flush request was accepted

3.4.1.7 MKx_RxAlloc()

A callback function invoked by the LLC to allocate a receive packet buffer.

```
typedef tMKxStatus(* fMKx_RxAlloc)(struct MKx *pMKx,
                                   int BufLen,
                                   uint8_t **ppBuf,
                                   void **ppPriv)
```

Parameters:

pMKx	**MKx** handle
BufLen	Maximum length of the receive packet
ppBuf	Pointer to a to-be-allocated buffer for the receive packet. In the case of an error: *ppBuf == NULL
ppPriv	Pointer to provide when invoking any callback associated with this receive packet. Usually the provided contents of ppBuf and ppPriv have

some association

Returns:

MKx_STATUS_SUCCESS if the receive packet allocation request was successful. Other values may be logged by the LLC for debug purposes.

Note:

The buffer must lie in DMA accessible memory. A possible implementation is shown below:

```
*ppPriv = alloc_skb(BufLen, GFP_DMA|GFP_ATOMIC);
*ppBuf = (*ppPriv)->data;
```

3.4.1.8 MKx_RxInd()

A callback function invoked by the LLC to deliver a receive packet buffer to the stack.

```
typedef tMKxStatus(* fMKx_RxInd)(struct MKx *pMKx,
                                 tMKxRxPacket *pRxPkt,
                                 void *pPriv)
```

Parameters:

pMKx **MKx** handle

pRxPkt Pointer to the receive packet. (same as *ppBuf provided in MKx_RxAlloc)

pPriv Private packet pointer (same as provided in **fMKx_RxAlloc**)

Returns:

MKx_STATUS_SUCCESS if the receive packet delivery was successful. Other values may be logged by the LLC for debug purposes.

3.4.1.9 MKx_NotifInd()

A callback function invoked by the LLC to notify the stack that one of the following has occurred:

- A radio has encountered a UTC boundary
- A particular radio channel configuration is now active
- A radio or channel has experienced an error

```
typedef tMKxStatus(* fMKx_NotifInd)(struct MKx *pMKx,
                                    tMKxNotif Notif)
```

Parameters:

pMKx **MKx** handle

Notif The notification

Returns:

MKX_STATUS_SUCCESS if the 'notif' was successfully handled. Other values are logged for debug purposes.

3.4.1.10 MKx_GetTSFReq()

A function invoked by the stack to request a read of the MAC TSF time at the time of calling.

```
typedef tMKxStatus (*fMKx_GetTSFReq) (struct MKx *pMKx)
```

Parameters:

pMKx **MKx** handle

Returns:

MKX_STATUS_SUCCESS if the GetTSF request was accepted

3.4.1.11 MKx_GetTSFInd()

A callback function invoked by the LLC to provide the stack with the read of the MAC TSF time at the time of calling.

```
typedef tMKxStatus (*fMKx_GetTSFInd) (struct MKx *pMKx, tMKxTSF TSF)
```

Parameters:

pMKx **MKx** handle
TSF **tMKxTSF** Current TSF value

Returns:

MKx_STATUS_SUCCESS if the receive packet delivery was successful. Other values may be logged by the LLC for debug purposes.

3.4.1.12 MKx_SetTSF()

In order to discipline and align the MAC TSF to UTC, two mechanisms are possible, either via GPS or via timing advertisements. This function supports both mechanisms.
If GPS is available and the 1PPS is connected to the SAF5x00, the stack must provide periodic updates of the UTC time corresponding to the last PPS from the GNSS module. The stack should ideally call this function (with pSetTSF->SetTSFData.Cmd == UTC_AT_1PPS) after each PPS event based on the time indicated in the NMEA string. It is essential that the function is called prior to the next PPS event.

When GPS is not available, UTC synchronisation can be achieved by the use of timing advertisements. The stack can call this function (with pSetTSF->SetTSFData.Cmd == UTC_AT_TSF), with the UTC time and TSF timestamps provided. Given that the MAC TSF timestamps include the local UTC offset from previous calls to MKx_SetTSF, the SAF5x00 firmware will treat each update as a delta, to be applied to the current local UTC offset.

```
typedef tMKxStatus(* fMKx_SetTSF)(struct MKx *pMKx, tMKxSetTSF *pSetTSF)
```

Set the **MKx** TSF at the next PPS.

Parameters:

pMKx **MKx** handle
pSetTSF **tMKxSetTSF** The Set TSF command which contains the counter value (Unix epoch value) to apply at the last PPS or the UTC time corresponding to the MAC TSF time.

Returns:

MKX_STATUS_SUCCESS (0) or a negative error code

3.4.1.13 MKx_TempCfg()

A function invoked by the stack to alter the temperature reading configuration (manual or I2C temperature sensors) and the temperature to transmit power calibration configuration. Transmit power calibration includes the temperature to transmit power offset relationship (specified as a slope for each antenna

```
typedef tMKxStatus (*fMKx_TempCfg) (struct MKx *pMKx,
                                    tMKxTempConfig *pCfg);
```

Parameters:

pMKx **MKx** handle

pCfg Pointer to the new temperature configuration to apply

Returns:

MKX_STATUS_SUCCESS if the 'TempCfg' was successfully handled. Other values are logged for debug purposes.

3.4.1.14 MKx_Temp()

A function invoked by the stack to return the current on-board temperatures read by the MKx radio.

```
typedef tMKxStatus (*fMKx_Temp) (struct MKx *pMKx,
                                 tMKxTemp *pTemp);
```

Parameters:

pMKx **MKx** handle

pTemp Pointer to the temperature measurements

Returns:

MKX_STATUS_SUCCESS if the temperature read was successfully handled. Other values are logged for debug purposes.

3.4.1.15 MKx_PowerDetCfg()

A function invoked by the stack to alter the power detector configuration, which is used to perform the calibration of the transmit power. The configuration includes the ability to select the power calibration off, power calibration to be temperature based only or select the power detector to be used for calibration. The configuration also includes the power detector to output power (dBm) relationship, together with extra compensation across temperature. Lastly it includes the set period between tx power detector reads. This function can be used as a read-only command, where the current configuration is returned to pCfg, if the PowerCalMode of the configuration data is invalid.

```
typedef tMKxStatus (*fMKx_PowerDetCfg) (struct MKx *pMKx,
                                        tMKxPowerDetConfig *pCfg);
```

Parameters:

pMKx **MKx** handle

pCfg Pointer to the new configuration to apply

Returns:

MKX_STATUS_SUCCESS if the 'PowerDetCfg' was successfully handled. Other values are logged for debug purposes.

3.4.1.16 MKx_RSSICalCfg()

A function invoked by the stack to alter the receive signal strength indicator (RSSI) calibration configuration, which is used to perform the RSSI calibration across temperature variation. The configuration includes the ability to turn theRSSI calibration off or on. The configuration also includes the temperature to RSSI slope relationship and the zero temperature intercept points for each 10MHz channel (168 to 184). The compensation to the measured RSSI = Intercept + Slope*Temperature (in deg C). This function can be used as a read-only command, where the current configuration is returned to pCfg, if the RSSICalMode of the configuration data is set to MKX_RSSI_CAL_GET.

```
typedef tMKxStatus (*fMKx_RSSICalCfg) (struct MKx *pMKx,
                                       tMKxRSSICalConfig *pCfg);
```

Parameters:

pMKx **MKx** handle
pCfg Pointer to the new configuration to apply

Returns:
MKX_STATUS_SUCCESS if the 'RSSICalCfg' was successfully handled. Other values are logged for debug purposes.

3.4.1.17 MKx_DebugReq()

A function invoked by the stack to deliver a debug buffer to the MKx. This is used for sending or requesting proprietary debug information.

```
typedef tMKxStatus (*fMKx_DebugReq) (struct MKx *pMKx, struct MKxIFMsg *pMsg)
```

Parameters:
pMKx **MKx** handle
pMsg **MKxIFMsg** Pointer to the buffer

Returns:
MKX_STATUS_SUCCESS if the buffer was sent successful. Other values may be logged by the MKx for debug purposes.

3.4.1.18 MKx_DebugInd()

A callback function invoked by the LLC to deliver a debug buffer to the stack, following when an fMKx_DebugReq call has been issued. Note that the debug buffer must be handled or copied in the callback.

```
typedef tMKxStatus (*fMKx_DebugInd) (struct MKx *pMKx, struct MKxIFMsg *pMsg)
```

Parameters:
pMKx **MKx** handle
pMsg **MKxIFMsg** Pointer to the buffer

Returns:
MKx_STATUS_SUCCESS if the receive packet allocation delivery was successful. Other values may be logged by the MKx for debug purposes.

3.4.2 User-space specific API functions and callbacks

3.4.2.1 MKx_Recv

The user-space API includes this function to allow it to be integrated with existing event loops, etc. This is not needed in kernel space since the events come from other sources (USB stack, etc).

```
tMKxStatus MKx_Recv(tMKx *pMKx)

Handle messages from the MKx interface.

Parameters:
      pMKx        MKx handle

Returns:
      MKX_STATUS_SUCCESS (0) or a negative error code
```

3.4.3 Type definitions

3.4.3.1 tMKxIFMsgType

The MKx accepts messages with of the following types. Note the tMKxIFMsgType field is managed by the LLC.

```
/// Types for the LLCRemote message transfers
typedef enum
{
  /// A null packet (to indicate no message data)
  MKXIF_NONE        = 0,
  /// A transmit packet (message data is @ref tMKxTxPacket)
  MKXIF_TXPACKET    = 1,
  /// A received packet (message data is @ref tMKxRxPacket)
  MKXIF_RXPACKET    = 2,
  /// New UTC Time (message data is @ref tMKxSetTSF)
  MKXIF_SET_TSF     = 3,
  /// Transmitted packet event (message data is @ref tMKxTxEventData)
  MKXIF_TXEVENT     = 4,
  /// Radio config for Radio A (message data is @ref tMKxRadioConfig)
  MKXIF_RADIOACFG   = 5,
  /// Radio config for Radio B (message data is @ref tMKxRadioConfig)
  MKXIF_RADIOBCFG   = 6,
  /// Radio A statistics (message data is @ref tMKxRadioStats)
  MKXIF_RADIOASTATS = 7,
  /// Radio B statistics (message data is @ref tMKxRadioStats)
  MKXIF_RADIOBSTATS = 8,
  /// Flush a single queue or all queueus (message data is @ref tMKxFlushQueue)
  MKXIF_FLUSHQ      = 9,
  /// A generic debug container.
  MKXIF_DEBUG       = 10,
```

```
  /// C2XSEC message (message data is @ref tMKxC2XSec)
  MKXIF_C2XSEC      = 11,
  /// Temperature config message (message data is @ref tMKxTempConfig)
  MKXIF_TEMPCFG     = 12,
  /// Temperature measurement message (message data is @ref tMKxTemp)
  MKXIF_TEMP        = 13,
  /// Power detector config message (message data is @ref tMKxPowerDetConfig)
  MKXIF_POWERDETCFG = 14,
  /// Read the current UTC Time (message data is @ref tMKxGetTSF)
  MKXIF_GET_TSF     = 15,
  /// RSSI Cal config message (message data is @ref tMKxRSSICalConfig)
  MKXIF_RSSICALCFG  = 16,
  /// Invalid message type, used for array dimensioning
  MKXIF_COUNT       = 17,
  /// Invalid message type, used for bounds checking
  MKXIF_MAX = MKXIF_COUNT - 1,
} eMKxIFMsgType;
/// @copydoc eMKxIFMsgType
typedef uint16_t tMKxIFMsgType;
```

3.4.3.2 tMKxRadio

The MKx implements dual MAC & PHY, these values select which one.

```
/// MKx Radio
typedef enum MKXRadio
{
  /// Selection of Radio A of the MKX
  MKX_RADIO_A = 0,
  /// Selection of Radio B of the MKX
  MKX_RADIO_B = 1,
  // ...
  /// Used for array dimensioning
  MKX_RADIO_COUNT,
  /// Used for bounds checking
  MKX_RADIO_MAX = MKX_RADIO_COUNT - 1,
} eMKxRadio;
/// @copydoc eMKxRadio
typedef uint8_t tMKxRadio;
```

3.4.3.3 tMKxChannel

Each MKx radio (MAC + PHY) supports two channel configurations, these values specify one.

```
/// MKx Channel
typedef enum MKXChannel
{
  /// Indicates Channel configuration 0 is selected
  MKX_CHANNEL_0 = 0,
```

```
  /// Indicates Channel configuration 1 is selected
  MKX_CHANNEL_1 = 1,
  // ...
  /// Used for array dimensioning
  MKX_CHANNEL_COUNT,
  /// Used for bounds checking
  MKX_CHANNEL_MAX = MKX_CHANNEL_COUNT - 1,
} eMKxChannel;
/// @copydoc eMKxChannel
typedef uint8_t tMKxChannel;
```

3.4.3.4 tMKxRadioMode

Each MKx radio (MAC + PHY) can operate continuously using one of the above two channel configurations or in channel switching mode (alternating between the two channel configurations). These values indicate the mode.

```
typedef enum MKXRadioMode
{
  /// Radio is off
  MKX_MODE_OFF = 0,
  /// Radio is using channel configuration MKX_CHANNEL_0 only
  MKX_MODE_CHANNEL_0 = 1,
  /// Radio is using channel configuration MKX_CHANNEL_1 only
  MKX_MODE_CHANNEL_1 = 2,
  /// Radio is channel switching between MKX_CHANNEL_0 & MKX_CHANNEL_1
  MKX_MODE_SWITCHED = 3,
} eRadioMode;
typedef uint8_t tMKxRadioMode;
```

3.4.3.5 tMKxTxQueue

Each MKx channel configuration can support five transmit queues, these values specify one.

```
/// Transmit queues
typedef enum MKxTxQueue
{
  MKX_TXQ_NON_QOS = 0, /// Non QoS (for WSAs etc.)
  MKX_TXQ_AC_VO = 1,   ///< Voice
  MKX_TXQ_AC_VI = 2,   ///< Video
  MKX_TXQ_AC_BE = 3,   ///< Best effort
  MKX_TXQ_AC_BK = 4,   /// Background
  /// For array dimensioning
  MKX_TXQ_COUNT,
  /// For bounds checking
  MKX_TXQ_MAX = MKX_TXQ_COUNT - 1,
} eMKxTxQueue;
/// @copydoc eMKxTxQueue
typedef uint8_t tMKxTxQueue;
```

3.4.3.6 tMKxFlushQueue

This is the message format for flushing the transmit queues for the specified RadioID and ChannelID.

```
/*
 * MKx FlushQueue message format
 */
typedef struct MKxFlushQueue
{
  /// Interface Message Header (reserved area for LLC usage)
  tMKxIFMsg Hdr;
  /// Indicate the radio that should be used (Radio A or Radio B)
  tMKxRadio RadioID;
  /// Indicate the channel for the selected radio
  tMKxChannel ChannelID;
  /// Queue selection to be flush (MKX_TXQ_COUNT for all)
  tMKxTxQueue TxQueue;
} __attribute__((__packed__)) tMKxFlushQueue;
```

3.4.3.7 tMKx

This type is a pointer to an MKx structure that encompasses the entire exported API: Configuration, Statistics, Functions, & Callbacks. It is returned by a call to MKx_Init (), and is required by all calls to the MKx_* API functions.

```
/// MKx LLC handle
typedef struct MKx
{
  /// 'Magic' value used as an indicator that the handle is valid
  uint32_t Magic;
  /// Private data reference (for the stack to store stuff)
  void *pPriv;
  /// State information (read only)
  struct {
    /// Statistics (read only)
    tMKxRadioStats Stats[MKX_RADIO_COUNT];
    /// Temperature measurements (read only)
    tMKxTempData Temp;
  } State;
  /// Configuration (read only)
  struct {
    /// Configuration (read only)
    const tMKxRadioConfig Radio[MKX_RADIO_COUNT];
  } Config;
  /// MKx API functions and callbacks
  struct {
    /// Stack -> LLC
    struct {
      fMKx_Config Config;
```

```
      fMKx_TxReq TxReq;
      fMKx_GetTSFReq GetTSFReq;
      fMKx_SetTSF SetTSF;
      fMKx_TxFlush TxFlush;
      fMKx_TempCfg TempCfg;
      fMKx_Temp Temp;
      fMKx_PowerDetCfg PowerDetCfg;
      fMKx_RSSICalCfg RSSICalCfg;
      fMKx_DebugReq DebugReq;
      fC2XSec_CommandReq C2XSecCmd;
    } Functions;
    /// LLC -> Stack
    struct {
      fMKx_TxCnf TxCnf;
      fMKx_RxAlloc RxAlloc;
      fMKx_RxInd RxInd;
      fMKx_NotifInd NotifInd;
      fMKx_DebugInd DebugInd;
      fMKx_GetTSFInd GetTSFInd;
      fC2XSec_ReponseInd C2XSecRsp;
    } Callbacks;
  } API;
} tMKx;
```

3.4.3.8 tMKxIFMsg

The MKx interface message descriptor. This is the top level header data structure for all messages sent to and from the MKx Radio. This header is managed by the LLC itself.

```
/// LLCRemote message (LLC managed header)
typedef struct MKxIFMsg
{
  /// Message type
  tMKxIFMsgType Type;
  /// Length of the message, including the header itself
  uint16_t Len;
  /// Sequence number
  uint16_t Seq;
  /// Return value
  int16_t Ret;
  /// Message data
  uint8_t Data[];
} __attribute__ ((packed)) tMKxIFMsg;
```

3.4.3.9 tMKxTxCtrlFlags

The MKx Tx Control flags field is used to signal special (non-standard) behaviour of the transmitter.

```
/// LLCRemote message (LLC managed header)
typedef enum
{
  /// Do not modify the sequence number field
  MKX_DISABLE_MAC_HEADER_UPDATES_SEQCTRL    = 0x01,
  /// Do not modify the duration ID field
  MKX_DISABLE_MAC_HEADER_UPDATES_DURATIONID = 0x02,
  /// Do not modify the Ack Policy field
  MKX_DISABLE_MAC_HEADER_UPDATES_ACKPOLICY  = 0x04,
  /// Do not modify the Retry field and set Max retries to zero
  MKX_DISABLE_MAC_HEADER_UPDATES_RETRY      = 0x08,
  /// Force the use of RTS/CTS with this packet
  MKX_FORCE_RTSCTS                          = 0x10,
} eMKxTxCtrlFlags;
typedef uint8_t tMKxTxCtrlFlags;
```

3.4.3.10 tMKxTxPacket

The MKx transmit descriptor and frame. This header is used to control how the data packet is transmitted by the LLC. This is the header used on all transmitted frames

```
/**
 * MKx Transmit Packet format.
 */
typedef struct MKxTxPacket
{
  /// Interface Message Header
  tMKxIFMsg Hdr;
  /// Tx Packet control and frame data
  tMKxTxPacketData TxPacketData;
} __attribute__((__packed__)) tMKxTxPacket;

/// MKx transmit descriptor
typedef struct MKxTxPacketData
{
  /// Indicate the radio that should be used (Radio A or Radio B)
  tMKxRadio RadioID;
  /// Indicate the channel config for the selected radio
  tMKxChannel ChannelID;
  /// Indicate the antennas upon which packet should be transmitted
  /// (may specify default)
  tMKxAntenna TxAntenna;
  /// Indicate the MCS to be used (may specify default)
  tMKxMCS MCS;
  /// Indicate the power to be used (may specify default)
  tMKxPower TxPower;
  /// Additional control over the transmitter behaviour (must be set to zero
```

```
  /// for normal operation)
  tMKxTxCtrlFlags TxCtrlFlags;
  // Reserved (for 64 bit alignment and internal processing)
  uint8_t Reserved0;
  /// Indicate the expiry time as an absolute MAC time in microseconds
  /// (0 means never)
  tMKxTSF Expiry;
  /// Length of the frame (802.11 Header + Body, not including FCS)
  uint16_t TxFrameLength;
  // Reserved (for 32 bit alignment and internal processing)
  uint16_t Reserved1;
  /// Frame (802.11 Header + Body, not including FCS)
  uint8_t TxFrame[];
} __attribute__((__packed__)) tMKxTxPacketData;
```

3.4.3.11 tMKxTxEvent

The MKx transmit event. This is the packet transmission event announcement made by the LLC.

```
/**
 * MKx Transmit Event format.
 */
typedef struct MKxTxEvent
{
  /// Interface Message Header
  tMKxIFMsg Hdr;
  /// Tx Event Data
  tMKxTxEventData TxEventData;
} __attribute__((__packed__)) tMKxTxEvent;
```

3.4.3.12 tMKxTxEventData

The MKx transmit event data. This is the structure contained within the packet transmission event announcement.

```
/**
 * Transmit Event Data. This is the structure of the data field for
 * MKxIFMsg messages of type TxEvent.
 */
typedef struct MKxTxEventData
{
  /// Transmit status (transmitted/retired), @ref eMKxStatus
  int16_t TxStatus;
  /// 802.11 MAC sequence number of the transmitted frame
  uint16_t MACSequenceNumber;
  /// The TSF when the packet was transmitted or retired
  tMKxTSF TxTime;
```

```
  /// Delay (VDSP ticks) between end of Tx Data frame and start of Rx Ack frame
  /// Note VDSP Clock runs at 300MHz
  uint32_t AckResponseDelay;
  /// Delay (VDSP ticks) between end of Tx RTS frame and start of Rx CTS frame
  /// Note VDSP Clock runs at 300MHz
  uint32_t CTSResponseDelay;
  /// Time (us) between the arrival of the packet at the MAC and its Tx
  uint32_t MACDwellTime;
  /// Short packet retry counter
  uint8_t NumShortRetries;
  /// Long packet retry counter
  uint8_t NumLongRetries;
  /// Destination address of the transmitted frame
  uint8_t DestAddress[6];
} __attribute__((__packed__)) tMKxTxEventData;
```

3.4.3.13 tMKxRxPacket

The MKx Receive descriptor and frame. This header is pre-pended to all received packets and is used to pass receive packet meta-information from the LLC to the stack.

```
/**
 * MKx receive packet format.
 */
typedef struct MKxRxPacket
{
  /// Interface Message Header
  tMKxIFMsg Hdr;
  /// Rx Packet control and frame data
  tMKxRxPacketData RxPacketData;
} __attribute__((__packed__)) tMKxRxPacket;

typedef struct MKxRxPacketData
{
  /// Indicate the radio that was used (Radio A or Radio B)
  tMKxRadio RadioID;
  /// Indicate the channel config for the selected radio
  tMKxChannel ChannelID;
  /// Indicate the data rate that was used
  tMKxMCS MCS;
  /// Indicates FCS passed for received frame (1=Pass, 0=Fail)
  uint8_t FCSPass;
  /// Indicate the received power on Antenna 1
  tMKxPower RxPower1;
  /// Indicate the received power on Antenna 2
  tMKxPower RxPower2;
  /// Indicate the receiver noise on Antenna 1
  tMKxPower RxNoise1;
```

```
  /// Indicate the receiver noise on Antenna 2
  tMKxPower RxNoise2;
  /// Per Frame Receive Meta Data
  tMKxRxMeta RxMeta;
  /// MAC Rx Timestamp, local MAC TSF time at which packet was received
  tMKxTSF RxTSF;
  /// Length of the Frame (802.11 Header + Body, not including FCS)
  uint16_t RxFrameLength;
  // Reserved (for 32 bit alignment)
  uint16_t Reserved1;
  /// Frame (802.11 Header + Body + FCS)
  uint8_t RxFrame[];
} __attribute__((__packed__)) tMKxRxPacketData;

/**
* MKx Meta Data type - contains per frame receive meta-data
*
* The frequency offset estimate is the measured offset with respect to the
* local oscillator frequency, which is accurate to +/- 10ppm.
*/
typedef struct MKxRxMeta
{
  /// Estimated frequency offset of rx frame in Hz (with respect to local freq)
  uint32_t FreqOffset;
} __attribute__((__packed__)) tMKxRxMeta;
```

If only a single receive power measure is required, then simply take the maximum power of Antenna 1 and 2.

3.4.3.14 tMkxRadioConfig

```
/**
 * MKx configuration message format.
 */
typedef struct MKxRadioConfig
{
  /// Interface Message Header
  tMKxIFMsg Hdr;
  /// Radio configuration data
  tMKxRadioConfigData RadioConfigData;
} __attribute__((__packed__)) tMKxRadioConfig;

/// MKx per radio configuration
typedef struct MKxRadioConfigData
{
  /// Operation mode of the radio
  tMKxRadioMode Mode;
  /// Reserved (for 32 bit alignment)
```

```
  uint16_t Reserved0;
  /// Channel Configurations for this radio
  tMKxChanConfig ChanConfig[MKX_CHANNEL_COUNT];
} tMKxRadioConfigData;

/// MKx channel configuration
typedef struct MKxChanConfig
{
  /// PHY specific config
  struct MKxChanConfigPHY PHY;
  /// MAC specific config
  struct MKxChanConfigMAC MAC;
  /// LLC (WMAC) specific config
  struct MKxChanConfigLLC LLC;
} __attribute__((__packed__)) tMKxChanConfig;

/// PHY specific config
typedef struct MKxChanConfigPHY
{
  /// The channel centre frequency that should be used e.g. 5000 + (5 * 172)
  tMKxChannelFreq ChannelFreq;
  /// Indicate if channel is 10 MHz or 20 MHz
  tMKxBandwidth Bandwidth;
  /// Default Transmit antenna configuration
  /// (can be overridden in @ref tMKxTxPacket)
  /// Antenna selection used for transmission of ACK/CTS
  tMKxAntenna TxAntenna;
  /// Receive antenna configuration
  tMKxAntenna RxAntenna;
  /// Indicate the default data rate that should be used
  tMKxMCS DefaultMCS;
  /// Indicate the default transmit power that should be used
  /// Power setting used for Transmission of ACK/CTS
  tMKxPower DefaultTxPower;
} __attribute__((__packed__)) tMKxChanConfigPHY;

/// MAC specific config
typedef struct MKxChanConfigMAC
{
  /// Dual Radio transmit control (inactive in single radio configurations)
  tMKxDualTxControl DualTxControl;
  /// The RSSI power detection threshold for carrier sense [dBm]
  int8_t CSThreshold;
  /// Slot time/duration, per 802.11-2007
  uint16_t SlotTime;
  /// Distributed interframe space, per 802.11-2007
  uint16_t DIFSTime;
```

```
  /// Short interframe space, per 802.11-2007
  uint16_t SIFSTime;
  /// Duration to wait after an erroneously received frame,
  /// before beginning slot periods
  /// @note this should be set to EIFS - DIFS
  uint16_t EIFSTime;
  /// Per queue configuration
  tMKxTxQConfig TxQueue[MKX_TXQ_COUNT];
  /// Address matching filters: DA, broadcast, unicast & multicast
  tMKxAddressMatching AMSTable[AMS_TABLE_COUNT];
  /// Retry limit for short unicast transmissions
  uint16_t ShortRetryLimit;
  /// Retry limit for long unicast transmissions
  uint16_t LongRetryLimit;
  /// Threshold at which RTS/CTS is used for unicast packets (bytes).
  uint16_t RTSCTSThreshold;
} __attribute__((__packed__)) tMKxChanConfigMAC;

/// LLC (WMAC) specific config
typedef struct MKxChanConfigLLC
{
  /// Duration of this channel interval, in microseconds. Zero means forever.
  uint32_t IntervalDuration;
  /// Duration of guard interval upon entering this channel, in microseconds
  uint32_t GuardDuration;
} __attribute__((__packed__)) tMKxChanConfigLLC;

/// MKx transmit queue configuration
typedef struct MKxTxQConfig
{
  /// Arbitration inter-frame-spacing (values of 0 to 16)
  uint8_t AIFS;
  /// Contention window min
  uint8_t CWMIN;
  /// Contention window max
  uint16_t CWMAX;
  /// TXOP duration limit [ms]
  uint16_t TXOP;
} tMKxTxQConfig;
```

3.4.3.15 tMKxRadioStats

The MKx reports statistics counters for each radio, channel configuration and transmit queue:

```
/// MKx Radio stats format
typedef struct MKxRadioStats
  /// Interface Message Header
```

```
  tMKxIFMsg Hdr;
  /// Radio Stats Data
  tMKxRadioStatsData RadioStatsData;
} __attribute__((__packed__)) tMKxRadioStats;

/// Radio level stats counters
typedef struct MKxRadioStatsData
{
  /// Per channel context statistics
  tMKxChannelStats Chan[MKX_CHANNEL_COUNT];
  /// TSF timer value at the end of the last measurement period [us]
  tMKxTSF TSF;
} __attribute__((__packed__)) tMKxRadioStatsData;

/// Channel stats counters
typedef struct MKxChannelStats
{
  /// Number of frames submitted via MKx_TxReq()
  uint32_t    TxReq;
  /// Number of Tx frames discarded by the MKx
  uint32_t    TxFail;
  /// Number of frames successfully transmitted (excluding retries)
  uint32_t    TxCnf;
  /// Number of frames un-successfully transmitted (excluding retries)
  uint32_t    TxErr;
  /// Number of packets transmitted on the channel (including retries and RTS,CTS
ACKs)
  uint32_t    TxValid;
  /// Number of frames delivered via MKx_RxInd()
  uint32_t    RxInd;
  /// Number of Rx frames discarded by the MKx
  uint32_t    RxFail;
  /// Total number of duplicate (unicast) packets received on the channel
  uint32_t    RxDup;
  /// Per queue statistics
  tMKxTxQueueStats TxQueue[MKX_TXQ_COUNT];
  /// Medium busy time.  Number of us that the medium is declared busy over
  /// the last measurement period.  Medium is declared busy during Tx, Rx and
  /// Nav events.
  uint32_t MediumBusyTime;
  /// Proportion of time which the radio is considered busy over the last
  /// measurement period. (255 = 100%)
  uint8_t ChannelBusyRatio;
  /// Average idle period power [dBm]
  int8_t AverageIdlePower;
} tMKxChannelStats;
```

```
/// Tx Queue stats counters
typedef struct MKxTxQueueStats
{
  /// Number of frames submitted via MKx_TxReq() to the current queue
  uint32_t    TxReqCount;
  /// Number of frames successfully transmitted (excluding retries)
  uint32_t    TxCnfCount;
  /// Number of frames un-successfully transmitted (excluding retries)
  uint32_t    TxErrCount;
  /// Number of packets transmitted on the channel (including retries and RTS,CTS
ACKs)
  uint32_t    TxValid;
  /// Number of packets in the queue
  uint32_t    TxPending;
} tMKxTxQueueStats;
```

3.4.3.16 tMKxAddressMatching

The MKx MACs provide a flexible address matching mechanism to support multiple receive addresses or membership to specific multicast groups. Each MKx channel supports a table of 8 address matching entries.

```
typedef struct MKxAddressMatching
{
  /// 48 bit mask to apply to DA before comapring with Addr field
  uint64_t Mask:48;
  uint64_t :0; // Align to 64 bit boundary

  /// 48 bit MAC address to match after masking
  uint64_t Addr:48;
  /// Bitmask see @ref eMKxAddressMatchingCtrl
  uint64_t MatchCtrl:4;
  uint64_t :0; // Align to 64 bit boundary
} tMKxAddressMatching;
```

MKx MAC general operation on 802.11 frame reception:

- bitwise AND of 'Mask' and the incoming frame's DA (DA not modified)
- equality check between 'Addr' and the masked DA
- If equal: continue
- If 'ResponseEnable' is set: Send 'ACK'
- If 'BufferEnableCtrl' is set: Copy into internal buffer & deliver via RxInd() if FCS check passes
- If 'BufferEnableBadFCS' is set: Deliver via RxInd() even if FCS check fails
-

To receive broadcast frames:

```
Addr = 0XFFFFFFFFFFFFULL
Mask = 0XFFFFFFFFFFFFULL
MatchCtrl = 0x0000
```

To receive anonymous IEEE1609 heartbeat (multicast) frames:

```
Addr = 0X000000000000ULL
Mask = 0XFFFFFFFFFFFFULL
MatchCtrl = 0x0000
```

To receive valid unicast frames for 01:23:45:67:89:AB (our MAC address)

```
Addr = 0XAB8967452301ULL
Mask = 0XFFFFFFFFFFFFULL
MatchCtrl = 0x0001
```

To monitor the channel in promiscious mode (including failed FCS frames and all duplicates):

```
Addr = 0X000000000000ULL
Mask = 0X000000000000ULL
MatchCtrl = 0x0016
```

3.4.3.17 eAddressMatchingCtrl

Address matching control bits allow for detailed control of the address matching of received frames.

```
/**
 * Address matching control bits
 * (bit 0) = ResponseEnable
 * (bit 1) = BufferEnableCtrl
 * (bit 2) = BufferEnableBadFCS
 * (bit 3) = LastEntry
 * (bit 4) = BufferDuplicate
 */
typedef enum
{
  /// ResponseEnable -- Respond with ACK when a DATA frame is matched.
  MKX_ADDRMATCH_RESPONSE_ENABLE = (1 << 0),
  /// BufferEnableCtrl -- Buffer control frames that match.
  MKX_ADDRMATCH_ENABLE_CTRL = (1 << 1),
  /// BufferEnableBadFCS -- Buffer frames even if FCS error was detected.
  MKX_ADDRMATCH_ENABLE_BAD_FCS = (1 << 2),
  /// LastEntry -- Indicates this is the last entry in the table.
  MKX_ADDRMATCH_LAST_ENTRY = (1 << 3),
  /// BufferDuplicate -- Buffer duplicate frames
  MKX_ADDRMATCH_DUPLICATE = (1 << 4),
} eMKxAddressMatchingCtrl;
```

3.4.3.18 tMKxBandwidth

10MHz or 20MHz channel configuration

```
/// MKx Bandwidth
typedef enum MKXBandwidth
{
  /// Indicates 10 MHz
  MKXBW_10MHz = 10,
  /// Indicates 20 MHz
  MKXBW_20MHz = 20,
} eMKxBandwidth;
```

```
/// @copydoc eMKxBandwidth
typedef int8_t tMKxBandwidth;
```

3.4.3.19 tMKxChannelFreq

The channel's centre frequency in MHz units.

```
typedef uint16_t tMKxChannelFreq;
```

For example:

- IEEE1609.4 CCH (178): 5890
- ETSI TC-ITS CCH (180): 5900
- 802.11a channel 60: 5300
- 802.11b/g channel 11: 2462
- ARIB STD-T109 (760MHz): 760

3.4.3.20 tMKxDualTxControl

When operating in a dual-radio configuration, it is necessary to consider the state of the other channel before transmission can be performed. Whilst the other radio is transmitting, the MAC may prevent the local radio from transmitting. In this case, when the other radio is transmitting, the local radio behaves as if the local channel is busy.

If the other channel is receiving, the LLC allows the following configuration options for the channel access function:

- **No consideration of other radio**. In this case, the radio will transmit without regard to whether the radio is currently receiving or not.
- **Rx inhibit.** In this mode, the MAC will prevent this radio from transmitting while the other radio is actively receiving a frame. In this case, when the other radio is receiving, the local radio behaves as if the local channel is busy. This prevents transmissions from this radio from corrupting the reception of a frame on the other radio, tuned to a nearby radio channel (in particular when shared or co-located antennas are in use).

In all cases, the transmission inhibit occurs at the MAC channel-access level, so packets will not be dropped when transmission is inhibited, they will simply be deferred.

```
/**
* MKx dual radio transmit control
* Controls transmit behaviour according to activity on the
* other radio (inactive in single radio configurations)
*/
typedef enum MKXDualTxControl
{
  /// Do not constrain transmissions
  MKX_TXC_NONE,
  /// Prevent transmissions when other radio is transmitting
  MKX_TXC_TX,
  /// Prevent transmissions when other radio is receiving
  MKX_TXC_RX,
```

```
  /// Prevent transmissions when other radio is transmitting or receiving
  MKX_TXC_TXRX,
  /// Default behaviour
  MKX_TXC_DEFAULT = MKX_TXC_TX,
} eMKxDualTxControl;
/// @copydoc eMKxDualTxControl
typedef uint8_t tMKxDualTxControl;
```

3.4.3.21 tMKxMCS

The modulation and coding scheme (MCS) or 'datarate'

```
typedef enum MKxMCS
{
  /// Rate 1/2 BPSK
  MKXMCS_R12BPSK = 0xB,
  /// Rate 3/4 BPSK
  MKXMCS_R34BPSK = 0xF,
  /// Rate 1/2 QPSK
  MKXMCS_R12QPSK = 0xA,
  /// Rate 3/4 QPSK
  MKXMCS_R34QPSK = 0xE,
  /// Rate 1/2 16QAM
  MKXMCS_R12QAM16 = 0x9,
  /// Rate 3/4 16QAM
  MKXMCS_R34QAM16 = 0xD,
  /// Rate 2/3 64QAM
  MKXMCS_R23QAM64 = 0x8,
  /// Rate 3/4 64QAM
  MKXMCS_R34QAM64 = 0xC,
  /// Use default data rate
  MKXMCS_DEFAULT = 0x0,
  /// Use transmit rate control
  MKXMCS_TRC = 0x1,
} eMKxMCS;
/// @copydoc eMKxMCS
typedef uint8_t tMKxMCS;
```

3.4.3.22 tMKxPower

The MKx PHY supports setting the transmit power on a per-packet basis and measuring the RSSI in half dB units. Certain values (shown below) are reserved and have special meaning.

```
/// Tx & Rx power of frame, in 0.5dB units.
typedef enum MKxPower
{
   /// Selects the PHY maximum transmit power
  MKX_POWER_TX_MAX     = INT16_MAX,
  /// Selects the PHY minimum transmit power
```

```
  MKX_POWER_TX_MIN     = INT16_MIN,
  /// Selects the PHY default transmit power level
  MKX_POWER_TX_DEFAULT = MKX_POWER_TX_MIN + 1,
} eMKxPower;
/// @copydoc eMKxPower
typedef int16_t tMKxPower;
```

3.4.3.23 tMKxAntenna

The MKx PHY supports maximum ratio combining on receive and cyclic delay diversity on transmit. Alternatively the channel configurations can be set to use a single specific antenna for reception and/or transmit. In addition the transmit antenna can be selected on a per-packet basis in the tMKxTxPacket header.

These values are used to select particular antennas.

```
/// Number of antennas present on the MKX
#define MKX_ANT_COUNT 2
/// MKx Antenna Selection
typedef enum MKxAntenna
{
  /// Transmit packet on neither antenna (dummy transmit)
  MKX_ANT_NONE = 0,
  /// Transmit packet on antenna 1
  MKX_ANT_1 = 1,
  /// Transmit packet on antenna 2 (when available).
  MKX_ANT_2 = 2,
  /// Transmit packet on both antenna
  MKX_ANT_1AND2 = MKX_ANT_1 | MKX_ANT_2,
  /// Selects the default (ChanConfig) transmit antenna setting
  MKX_ANT_DEFAULT = 4,
} eMKxAntenna;
/// @copydoc eMKxAntenna
typedef uint8_t tMKxAntenna;
```

3.4.3.24 tMKxNotif

Notification values signalled to the stack via the MKx_NotifInd() callback.

```
typedef enum MKxNotif
{
  MKX_NOTIF_NONE            = 0x0000000, ///< No notification
  // Useful masks
  MKX_NOTIF_MASK_ERROR      = 0x8000000, ///< Error
  MKX_NOTIF_MASK_UTC        = 0x4000000, ///< UTC boundary (PPS)
  MKX_NOTIF_MASK_STATS      = 0x2000000, ///< Statistics updated
  MKX_NOTIF_MASK_ACTIVE     = 0x1000000, ///< Radio channel active
  MKX_NOTIF_MASK_RADIOA     = 0x0000010, ///< Specific to radio A
  MKX_NOTIF_MASK_RADIOB     = 0x0000020, ///< Specific to radio B
```

```
  MKX_NOTIF_MASK_CHANNEL0    = 0x0000001, ///< Specific to channel 0
  MKX_NOTIF_MASK_CHANNEL1    = 0x0000002, ///< Specific to channel 1
  MKX_NOTIF_MASK_TEMPCFG     = 0x0000040, ///< Temperature configuration
  MKX_NOTIF_MASK_TEMP        = 0x0000080, ///< Temperature measurement
  MKX_NOTIF_MASK_POWERDETCFG = 0x0000100, ///< Power detector configuration
  /// Active: Radio A, Channel 0
  MKX_NOTIF_ACTIVE_A0      = MKX_NOTIF_MASK_ACTIVE | MKX_NOTIF_MASK_RADIOA |
                             MKX_NOTIF_MASK_CHANNEL0,
  /// Active: Radio A, Channel 1
  MKX_NOTIF_ACTIVE_A1      = MKX_NOTIF_MASK_ACTIVE | MKX_NOTIF_MASK_RADIOA |
                             MKX_NOTIF_MASK_CHANNEL1,
  /// Active: Radio B, Channel 0
  MKX_NOTIF_ACTIVE_B0      = MKX_NOTIF_MASK_ACTIVE | MKX_NOTIF_MASK_RADIOB |
                             MKX_NOTIF_MASK_CHANNEL0,
  /// Active: Radio B, Channel 1
  MKX_NOTIF_ACTIVE_B1      = MKX_NOTIF_MASK_ACTIVE | MKX_NOTIF_MASK_RADIOB |
                             MKX_NOTIF_MASK_CHANNEL1,
  /// Stats updated: Radio A, Channel 0
  MKX_NOTIF_STATS_A0       = MKX_NOTIF_MASK_STATS | MKX_NOTIF_MASK_RADIOA |
                             MKX_NOTIF_MASK_CHANNEL0,
  /// Stats updated: Radio A, Channel 1
  MKX_NOTIF_STATS_A1       = MKX_NOTIF_MASK_STATS | MKX_NOTIF_MASK_RADIOA |
                             MKX_NOTIF_MASK_CHANNEL1,
  /// Stats updated: Radio B, Channel 0
  MKX_NOTIF_STATS_B0       = MKX_NOTIF_MASK_STATS | MKX_NOTIF_MASK_RADIOB |
                             MKX_NOTIF_MASK_CHANNEL0,
  /// Stats updated: Radio B, Channel 1
  MKX_NOTIF_STATS_B1       = MKX_NOTIF_MASK_STATS | MKX_NOTIF_MASK_RADIOB |
                             MKX_NOTIF_MASK_CHANNEL1,
  /// UTC second boundary
  MKX_NOTIF_UTC            = MKX_NOTIF_MASK_UTC,
  /// Temperature configuration update
  MKX_NOTIF_TEMPCFG        = MKX_NOTIF_MASK_TEMPCFG,
  /// Temperature measurement update
  MKX_NOTIF_TEMP           = MKX_NOTIF_MASK_TEMP,
  /// Power detector configuration update
  MKX_NOTIF_POWERDETCFG    = MKX_NOTIF_MASK_POWERDETCFG,
  /// Error
  MKX_NOTIF_ERROR          = MKX_NOTIF_MASK_ERROR,
} eMKxNotif;
typedef uint32_t tMKxNotif;
```

3.4.3.25 tMKxStatus

Return code of all of the API function calls.

```
/// MKx MLME interface return codes
typedef enum
```

```
{
  /// Success return code
  MKXSTATUS_SUCCESS = 0,
  // -1 to -255 reserved for @c errno values (see <errno.h>)
  /// Unspecified failure return code (catch-all)
  MKXSTATUS_FAILURE_INTERNAL_ERROR              = -256,
  ...
  /// For all other return codes, please see MKxAPI.h
} eMKxStatus;
/// @copydoc eMKxStatus
typedef int tMKxStatus;
```

3.4.3.26 tMKxTempSource

The source of the temperature values can be one of the following.

```
/// MKx temperature sensor source
typedef enum MKXRadio
{
  /// No I2C sensors present, temperatures set via the MKXIF_TEMP command
  MKX_TEMP_SOURCE_MANUAL = 0,
  /// Single I2C sensor, acting for both PAAnt1 and PAAnt2 temperature settings
  MKX_TEMP_SOURCE_PA1_ONLY = 1,
  /// Dual I2C sensors, one for each PA (PAAnt1, PAAnt2)
  MKX_TEMP_SOURCE_BOTH = 2
} eMKxTempSource;
/// @copydoc eMKxTempSource
typedef uint8_t tMKxTempSource;
```

3.4.3.27 tMKxTempConfig

Temperature configuration data structure, specifying:

- How the temperatures are supplied to the MKx radio (either manually or via on-board I2C temperature sensors).
- I2C temperature sensor addresses.
- Transmit power temperature compensation configuration, where the transmit power offset to be applied to each requested transmit power is specified as an offset and a factor (slope), where the slope is dependent on the temperature (i.e. Tx Power Offset = PowerCalTempOffsetAntx + PowerCalTempFactorAntx * Temperature)
- Transmit power limitation configuration, where the transmit power is limited when a temperature threshold is met.

```
/**
 * Temperature Config request/indication (see @ref tMKxTempConfigData)
 */
typedef struct MKXTempConfig
{
  /// Interface Message Header (reserved area for LLC usage)
  tMKxIFMsg Hdr;
```

```
  /// Temperature Config Data
  tMKxTempConfigData TempConfigData;
}

/// MKx temperature sensor source
typedef struct MKXTempConfigData
{
  /// Number of I2C temperature sensors connected to the SAF5x00
  tMKxTempSource SensorSource;
  /// I2C Address for the PA Ant1 I2C temperature sensor
  uint8_t I2CAddrPAAnt1;
  /// I2C Address for the PA Ant2 I2C temperature sensor
  uint8_t I2CAddrPAAnt2;
  /// Power calibration factor for temperature compensation (S15Q16 format)
  int32_t PowerCalTempFactorAnt1;
  /// Power calibration offset (dB) for temperature compensation (S15Q16 format)
  int32_t PowerCalTempOffsetAnt1;
  /// Power calibration factor for temperature compensation (S15Q16 format)
  int32_t PowerCalTempFactorAnt2;
  /// Power calibration offset (dB) for temperature compensation (S15Q16 format)
  int32_t PowerCalTempOffsetAnt2;
  /// Maximum temp (degC) for when tx power is limited to PowerLimitMaxPower
  int16_t PowerLimitMaxTemp;
  /// Set maximum power (in 0.5 dBm units) when maximum temperature is reached
  tMKxPower PowerLimitMaxPower;
} __attribute__((__packed__)) tMKxTempConfigData;
```

3.4.3.28 tMKxTemp

Temperature read data structure, containing temperature values read close to antenna 1 PA and antenna 2 PA.

```
/**
 * Temperature measurement message
 * Used to indicate (or manually set) the two temperatures used for tx power
 * compensation.
 * @note This message is only accepted by the SAF5x00 when SensorSource in
 * MKxTempConfig is set to MKX_TEMP_SOURCE_MANUAL.
 */
typedef struct MKxTemp
{
  /// Interface Message Header (reserved area for LLC usage)
  tMKxIFMsg Hdr;
  /// Temperature measurement data
  tMKxTempData TempData;
} __attribute__((__packed__)) tMKxTemp;

/// MKx temperature sensor source
```

```
typedef struct MKxTempData
{
  /// Temperature setting in degrees C for PA Ant1, when no I2C sensors present
  int8_t TempPAAnt1;
  /// Temperature setting in degrees C for PA Ant2, when no I2C sensors present
  int8_t TempPAAnt2;
} __attribute__((__packed__)) tMKxTempData;
```

3.4.3.29 tMKxPowerDetConfig

Power detector configuration data structure, specifying:

- Top level transmit power calibration mode (select between off, temperature only, power detector). Note when power detector is selected, the radio will rely on temperature power calibration until the first valid (above the 1st calibration point) power detection read is performed. When this happens, the transmit power calibration always relies on the power offset as determined by the power detector.
- Power detector calibration data structure which consists of two lines which model the power detector vs output power relationship. Each line is specified via a single point (power detector value vs output power) and a rate from that line. The 1st calibration point specifies the minimum power detector value for the antenna. Any power detector values below this are ignored. The data structure also includes compensation due to temperature. This is specified by a temperature value which corresponds to the temperature that the power detector vs output power was calibrated to and then a rate at which the power is adjusted due to temperature.
- Period between tx power detector reads. Note radio will only perform a power detector read on a single antenna. Thus when dual antenna transmissions are enabled, the radio toggles between each antenna, such that the next power detector read enabled will be performed on the other antenna.

```
/**
 * Power detector config request/indication (see @ref tMKxPowerDetConfigData)
 */
typedef struct MKxPowerDetConfig
{
  /// Interface Message Header (reserved area for LLC usage)
  tMKxIFMsg Hdr;
  /// Power detector Config Data
  tMKxPowerDetConfigData PowerDetConfigData;
} __attribute__((__packed__)) tMKxPowerDetConfig;

/**
 * Power detector config data
 * Config data for the power detector, includes
 * - mode switch, so the power calibration can be disabled/changed.
 * - power detector calibration data for both antennas
 * - power detector read period
 */
typedef struct MKxPowerDetConfigData
```

```
{
  /// Number of I2C temperature sensors connected to the SAF5x00
  tMKxPowerCalMode PowerCalMode;
  /// Reserved (for 32 bit alignment)
  uint8_t Reserved0;
  uint8_t Reserved1;
  uint8_t Reserved2;
  /// Power detector calibration data for both antennas
  tMKxPowerDetCalData Cal;
  /// Period between enabling the tx power detector reads, in us
  uint32_t PowerDetReadPeriod;
} __attribute__((__packed__)) tMKxPowerDetConfigData;

/**
 * Power detector calibration data
 * Used to configure the tx power detector calibration by specifying the
 * TxPowerDet relationship to actual/measured power.  Specification is in the
 * form of two lines (specified as two points and two lines).
 * All TxPowerDet values below the 1st calibration point are ignored.
 */
typedef struct MKxPowerDetCalData
{
  /// Power detector 1st calibration point/line (Ant1)
  tMKxPowerDetCalPoint CalPoint1Ant1;
  /// Power detector 2nd calibration point/line (Ant1)
  tMKxPowerDetCalPoint CalPoint2Ant1;
  /// Power detector 1st calibration point/line (Ant2)
  tMKxPowerDetCalPoint CalPoint1Ant2;
  /// Power detector 2nd calibration point/line (Ant2)
  tMKxPowerDetCalPoint CalPoint2Ant2;
  /// Power detector temperature calibration point/line
  tMKxPowerDetTempCalPoint TempCalPoint;
} __attribute__((__packed__)) tMKxPowerDetCalData;

/**
 * Data structure that defines a temperature offset adjustment line in the Tx
 * power offset vs temperature relationship.
 * The rate parameter is in dBm per degree Celcius.
 */
typedef struct MKxPowerDetTempCalPoint
{
  /// Power detector calibration temperature
  /// Temperature where the power detector curves have been calculated at
  int32_t CalTemp;
  /// Power offset rate dBm/deg C (S15Q16)
  int32_t TempOffsetRate;
```

```
} __attribute__((__packed__)) tMKxPowerDetTempCalPoint;

/**
 * Data structure that defines a calibration point and line in the Tx power
 * detector value vs actual transmit power relationship.
 * The rate parameter is in dBm per PowerDetValue
 */
typedef struct MKxPowerDetCalPoint
{
  /// Power detector calibration point, power detector value
  int32_t PowerDet;
  /// Power detector calibration point, power in dBm value (S15Q16 format)
  int32_t PowerConstant;
  /// Power detector calibration rate dBm/PowerDet from defined point (S15Q16)
  int32_t PowerRate;
} __attribute__((__packed__)) tMKxPowerDetCalPoint;
/// MKx power calibration mode selection
typedef enum MKxPowerCalMode
{
  /// No tx power calibration
  MKX_POWER_CAL_OFF = 0,
  /// Use only the temperature for tx power calibration
  MKX_POWER_CAL_TEMP_ONLY = 1,
  /// Use the tx power detector for calibration.  Note this mode uses
  /// temperature based power calibration until 1st valid TxPowerDet read.
  MKX_POWER_CAL_POWERDET = 2
} eMKxPowerCalMode;
/// @copydoc eMKxPowerCalMode
typedef uint8_t tMKxPowerCalMode;
```

3.4.3.30 tMKxRSSICalConfig

RSSI configuration data structure, specifying:

- Top level RSSI calibration mode (select between off, on, and a get command).
- RSSI calibration data structure which consists of a basic first order model of the temperature to RSSI relationship, with an additional offset dependency due to the channel frequency included. The zero temperature intercept points are specified for each supported channel, range 168 and 184 inclusive. The measured RSSI is corrected by adding a compensation value = FreqDependentIntercept[ChannelIndex] + Slope * CurrentTemperature. Note both Slope and FreqDependentIntercept parameters are signed 32 bit integers, in S15Q16 fixed point format. This means a value of 1.0 dBm = 2^16 = 65536.

```
/**
 * RSSI Cal config request/indication (see @ref tMKxRSSICalConfigData)
 */
typedef struct MKxRSSICalConfig
{
```

```
  /// Interface Message Header (reserved area for LLC usage)
  tMKxIFMsg Hdr;
  /// RSSI Calibration Config Data
  tMKxTempConfigData PowerDetConfigData;
} __attribute__((__packed__)) tMKxRSSICalConfig;

/**
 * RSSI Calibration config data
 * Config data for the RSSI compensation, includes
 * - mode switch, so the RSSI compensation can be disabled/changed.
 * - RSSI calibration data, where measured RSSI is adjusted by the
 *   compensation formula: FreqDependentIntercept + Slope*CurrentTemperature
 */
typedef struct MKxRSSICalConfigData
{
  /// Number of I2C temperature sensors connected to the SAF5x00
  tMKxRSSICalMode RSSICalMode;
  /// Reserved (for 32 bit alignment)
  uint8_t Reserved0;
  uint8_t Reserved1;
  uint8_t Reserved2;
  /// Slope for temperature compensation (S15Q16 format)
  int32_t Slope;
  /// Frequency dependent zero temperature intercept (S15Q16 format)
  int32_t FreqDependentIntercept[RSSI_CAL_CHANNEL_CNT];
} __attribute__((__packed__)) tMKxRSSICalConfigData;

/// MKx RSSI calibration operating mode
typedef enum MKxRSSICalMode
{
  /// No compensation
  MKX_RSSI_CAL_OFF = 0,
  /// RSSI compensation enabled
  MKX_RSSI_CAL_ON = 1,
  /// Get RSSI compensation
  MKX_RSSI_CAL_GET = 2
} eMKxRSSICalMode;
/// @copydoc eMKxRSSICalMode
typedef uint8_t tMKxRSSICalMode;
```

3.4.3.31 tMKxSetTSF

The message format for setting (synchronising) the MAC TSF to UTC time. There are two message command types, as detailed in eMKxSetTSFCmd.

- Stack to provide the UTC time that corresponds to the last 1PPS event
- Stack to provide the UTC time that corresponds to a TSF timestamp, based upon receiving a timing advertisement.

```
/// MKx SetTSF command type
typedef enum MKxSetTSFCmd {
  /// UTC time provided corresponds to the UTC time at 1PPS event
  UTC_AT_1PPS = 1,
  /// UTC time provided corresponds to the TSF timestamp provided
  UTC_AT_TSF = 2
} eMKxSetTSFCmd;
/// @copydoc eSetTSFCmd
typedef uint8_t tMKxSetTSFCmd;

/**
 * Set TSF data
 * Data for setting the time synchronisation function (TSF) to UTC time.
 * The TSF can be set to
 * - UTC time at GPS 1PPS (obtained from NMEA data)
 * - UTC time at TSF Timestamp (through the use of timing advertisements)
 */
typedef struct MKxSetTSFData
{
  /// Selects the UTC to with 1PPS or TSF Timestamp
  tMKxSetTSFCmd Cmd;
  // Reserved (for 32 bit alignment)
  uint8_t Reserved0;
  uint8_t Reserved1;
  uint8_t Reserved2;
  /// UTC Time at either previous 1PPS event or at TSF Timestamp
  tMKxTSF UTC;
  /// TSF Timestamp at UTC
  tMKxTSF TSF;
} __attribute__((__packed__)) tMKxSetTSFData;

/*
 * MKx Set TSF message format
 */
typedef struct MKxSetTSF
{
  /// Interface Message Header (reserved area for LLC usage)
  tMKxIFMsg Hdr;
  /// SetTSF Message Data
  tMKxSetTSFData SetTSFData;
} __attribute__((__packed__)) tMKxSetTSF;
```

3.4.3.32 tMKxGetTSF

The message format for reading the current MAC TSF value.

```
typedef struct MKxGetTSF
```

```
{
  /// Interface Message Header (reserved area for LLC usage)
  tMKxIFMsg Hdr;
  /// GetTSF Message Data (current TSF value)
  tMKxTSF TSF;
} __attribute__((__packed__)) tMKxGetTSF;
```

4 Usage Overview

This section provides an overview of the API usage, in order to clarify its functionality by a 3rd party stack.

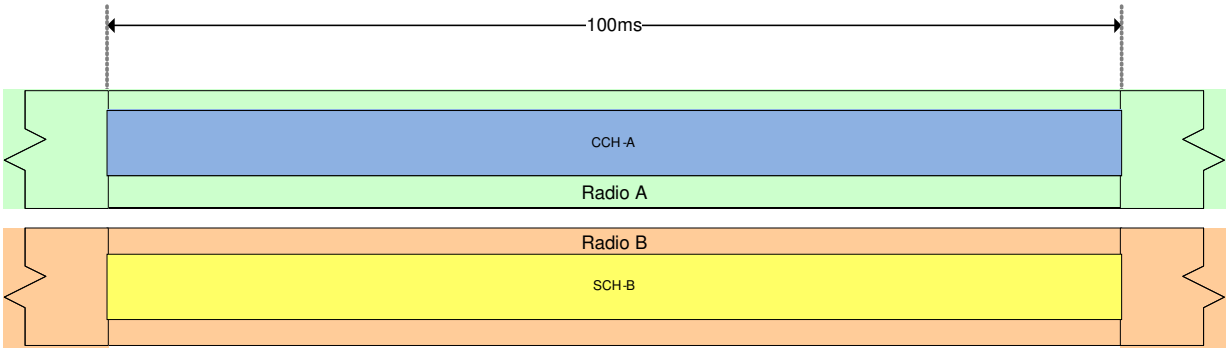
4.1 Radio setup scenarios

Typical single and dual radio IEEE1609 configurations are described in the following scenarios

4.1.1 Continuous dual radio (CCH-A + SCH-B)

This setup consists of:

- A single channel configuration for each radio:

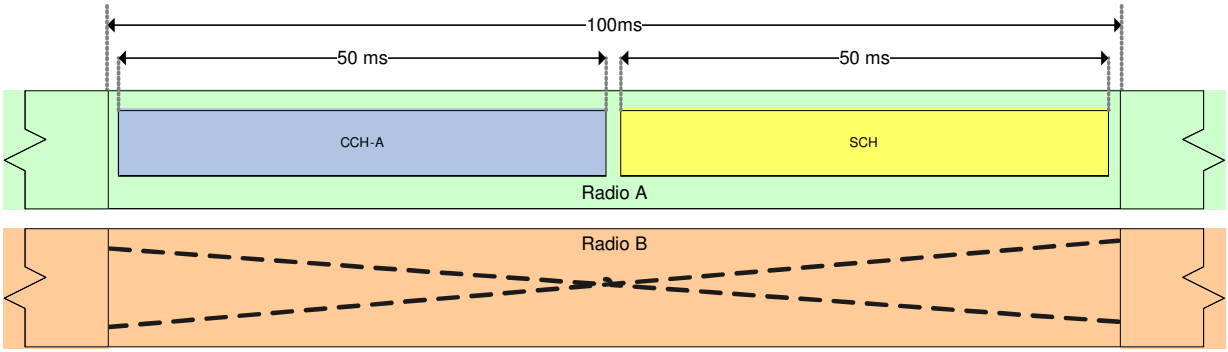


1. The stack populates the MKx_TxCnf, MKx_NotifInd, MKx_RxAlloc and MKx_RxInd callbacks in the MKx handle with its own functions
2. The stack reads radio A's tMKxChanneConfig values into a private copy
3. It modifies this structure for CCH-A (MKX_RADIO_A, MKX_CHANNEL_0)
4. It sets the mode parameter to MKX_MODE_CHANNEL_0
5. It applies this configuration using MKx_Config()
6. The stack reads radio B's tMKxChanneConfig values into a private copy
7. It modifies this structure for SCH-B (MKX_RADIO_B, MKX_CHANNEL_0)
8. It sets the mode parameter to MKX_MODE_CHANNEL_0
9. It applies this configuration using MKx_Config()

4.1.2 Alternating single radio (CCH-A/SCH channel switching)

This setup consists of:

- Two channel configurations active for a single radio.

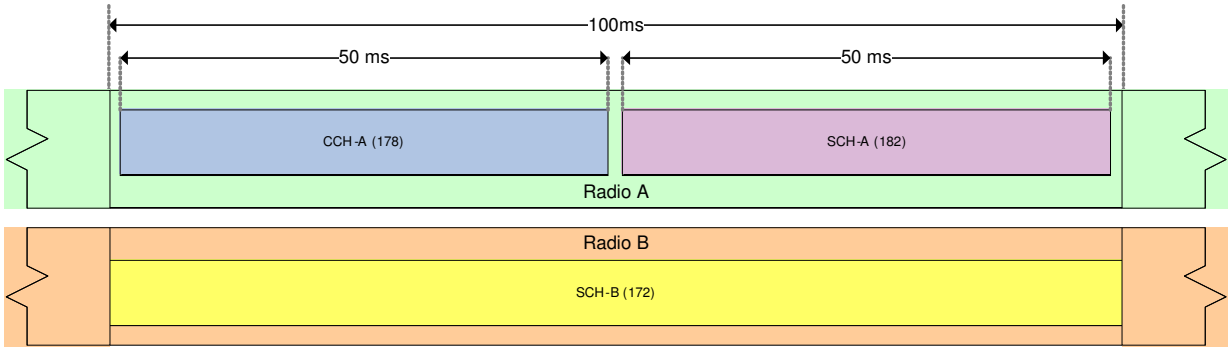


1. The stack populates the MKx_TxCnf, MKx_NotifInd, MKx_RxAlloc and MKx_RxInd callbacks in the MKx handle with its own functions
2. The stack reads radio A's tMKxChanneConfig values into a private copy
3. It modifies this structure for CCH-A (MKX_RADIO_A, MKX_CHANNEL_0)
4. It modifies this structure for SCH-A (MKX_RADIO_A, MKX_CHANNEL_1)
5. It sets the mode parameter to MKX_MODE_SWITCHED
6. It applies this configuration using MKx_Config()

4.1.3 Multichannel configuration (CCH-A/SCH-A + SCH-B)

This setup consists of:

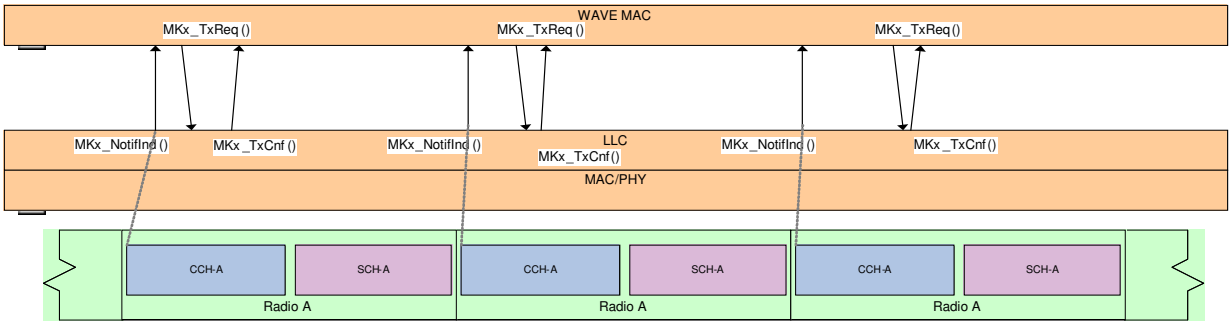
- Two channel configurations active on radio A
- A channel single configuration active on radio B



1. The stack populates the MKx_TxCnf, MKx_NotifInd, MKx_RxAlloc and MKx_RxInd callbacks in the MKx handle with its own functions
2. The stack reads radio A's tMKxChanneConfig values into a private copy
3. It modifies this structure for CCH-A (MKX_RADIO_A, MKX_CHANNEL_0)
4. It modifies this structure for SCH-A (MKX_RADIO_A, MKX_CHANNEL_1)
5. It sets the mode parameter to MKX_MODE_SWITCHED
6. It applies this configuration using MKx_Config()
7. The stack reads radio B's tMKxChanneConfig values into a private copy
8. It modifies this structure for SCH-B (MKX_RADIO_B, MKX_CHANNEL_0)
9. It sets the mode parameter to MKX_MODE_CHANNEL_0
10. It applies this configuration using MKx_Config()

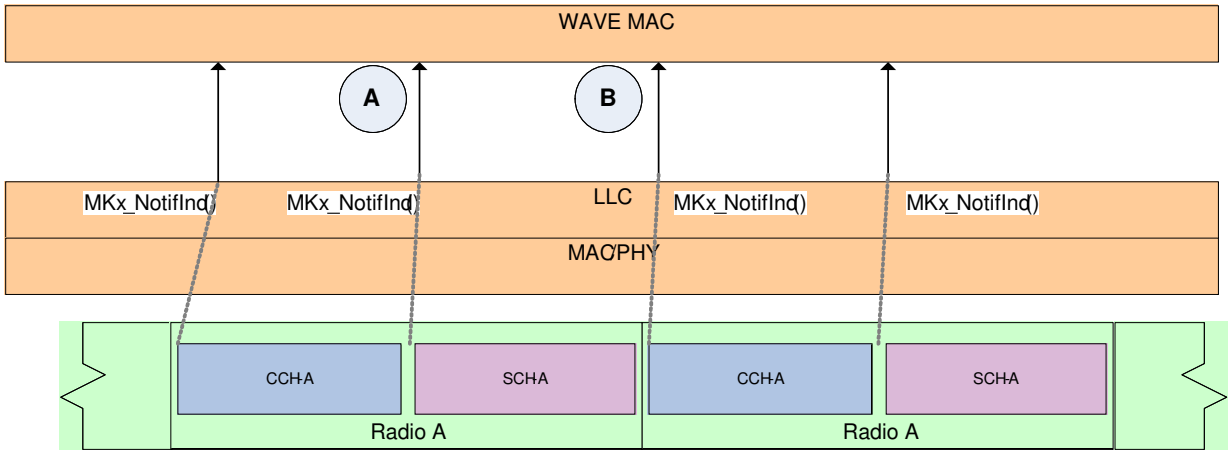
4.2 Context change notifications

The LLC provides notifications of context change events to enable the stack or applications to schedule specific transmissions. In this scenario, the stack broadcasts the periodic heartbeat frames with a random offset within the CCH-A interval.



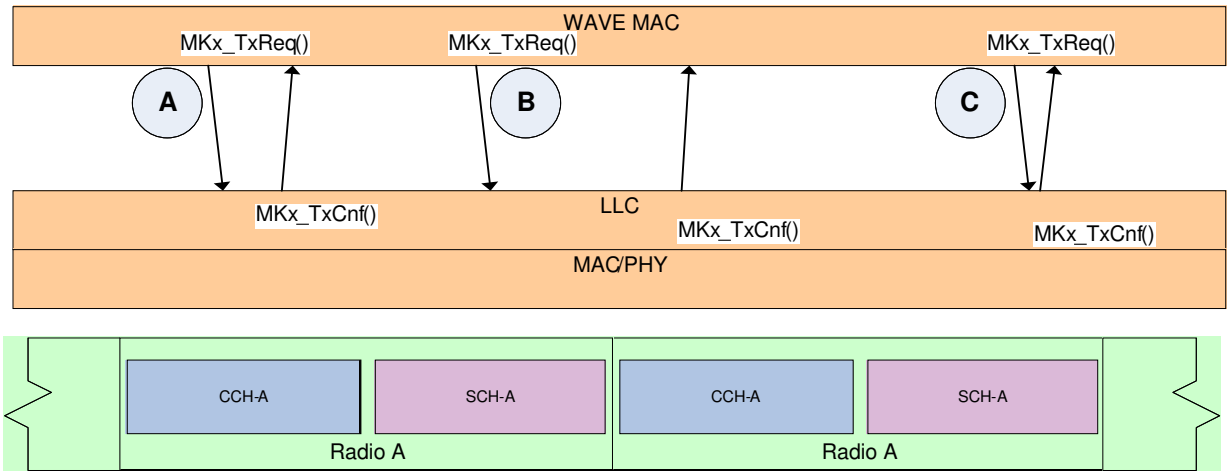
1. The CCH-A and SCH-A configurations are activated.
2. The LLC notifies the stack using MKx_NotifInd() that CCH-A has become active.
3. The stack performs a random delay (<50ms)
4. The stack submits a heartbeat frame for transmission using MKx_TxReq()
5. The LLC notifies the stack using MKx_TxCnf() that the frame was transmitted.

4.3 Statistics & measurement updates (every 50ms)



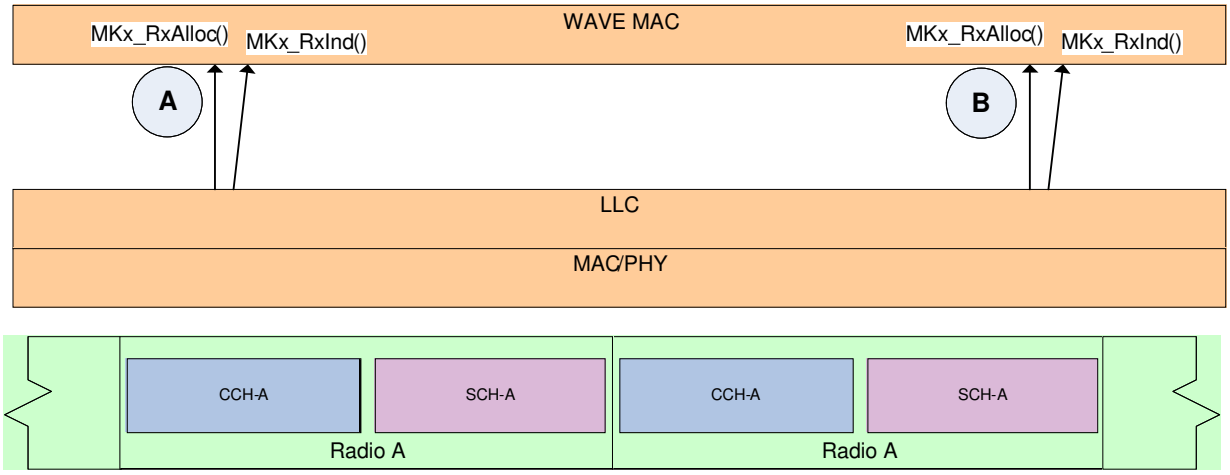
1. The CCH-A and SCH-A configurations are activated.
2. The LLC collects statistics and measurements from the underlying MAC when the CCH-A interval ends (A)
3. The LLC notifies the stack using MKx_NotifInd() that SCH-A has become active.
4. The stack can now inspect the latest CCH-A statistics counters and interval measurements
5. The LLC collects statistics and measurements from the underlying MAC when the SCH-A interval ends (B)
6. The LLC notifies the stack using MKx_NotifInd() that CCH-A has become active.
7. The stack can now inspect the latest SCH-A statistics counters and interval measurements

4.4 Transmit frame transfer



1. The CCH-A and SCH-A configurations are activated.
2. The stack submits a frame (A) for transmission on CCH-A using MKx_TxReq()
3. The MAC transmits the frame over the air in the CCH-A interval.
4. The LLC notifies the stack using MKx_TxCnf() that the frame was transmitted.
5. The STACK submits a frame (B) for transmission on CCH-A using MKx_TxReq()
6. Unfortunately, at that moment, the SCH-A interval is inactive. The MAC queues the frame but doesn't transmit it until the CCH-A interval becomes active again.
7. The LLC notifies the stack using MKx_TxCnf() that the frame was transmitted
8. The stack submits a frame (A) for transmission on CCH-A using MKx_TxReq()
9. The MAC transmits the frame over the air in the CCH-A interval.
10. The LLC notifies the stack using MKx_TxCnf() that the frame was transmitted.

4.5 Receive frame transfer



1. The CCH-A and SCH-A configurations are activated.
2. The MAC receives a frame (A) over the air in the CCH-A interval.
3. The LLC requests buffer allocations from the stack using MKx_RxAlloc() and the stack replies with the allocated memory pointers
4. The LLC DMA copies the receive frame into the provided buffers
5. The LLC notifies the stack using either the MKx handle's MKx_RxInd() callback
    - The radio and channel configuration values indicating CCH-A are populated in the tMKxRxPacket header
6. The MAC receives a frame (B) over the air in the SCH-A interval.

7. The LLC requests buffer allocations from the stack using MKx_RxAlloc() and the stack replies with the allocated memory pointers
8. The LLC DMA copies the receive frame into the provided buffers
9. The LLC notifies the stack using either the MKx handle's MKx_RxInd() callback
    - The radio and channel configuration values indicating SCH-A are populated in the tMKxRxPacket header

5 Security Variant

The SAF5x00EL/V110 variant supports the following acceleration operations at the expense of the second radio:

- Elliptic curve digital signature algorithm (ECDSA) acceleration, which provides security verifications of received messages
- Decompress public key operation
- Reconstruct public key operation

5.1 API

This section describes the API for the ECDSA accelerator.

5.1.1 API Functions & Callbacks

The following sections outline the LLC API functions and callbacks. Note that the usage of 'stack' below indicates the object using the API. These API functions provide a mechanism to transport commands & associated responses as defined in the NXP C2X Security API document [1].

5.1.1.1 fC2XSec_CommandReq()

A function invoked by the stack to deliver a C2X APDU buffer to the SAF5x00, for processing security commands. This function blocks until the buffer is sent on-the-wire

```
typedef tMKxStatus (*fC2XSec_CommandReq) (struct MKx *pMKx,
                                          tMKxC2XSec *pMsg);
```

Parameters:

pMKx **MKx** handle
pMsg Pointer to the buffer

Returns:

MKXSTATUS_SUCCESS if the buffer was sent successful. Other values may be logged by the MKx for debug purposes.

5.1.1.2 fC2XSec_ReponseInd()

A callback invoked by the LLC to deliver a C2X APDU buffer to the stack. pBuf must be handled (or copied) in the callback. Note that the ECDSA algorithm which runs inside the SAF5x00EL/V110 uses an internal FIFO to buffer received messages. If the FIFO becomes full, the security command message is returned in full by the SAF5x00 and LLC with the return code of the USB header set to -1. Thus it is necessary to check pMsg->Hdr.Ret for return of -1 in this callback function.

```
typedef tMKxStatus (*fC2XSec_ReponseInd) (struct MKx *pMKx,
                                          tMKxC2XSec *pMsg);
```

Parameters:

pMKx **MKx** handle
pMsg Pointer to the buffer

Returns:

```
MKXSTATUS_SUCCESS if the receive packet allocation delivery was successful.
Other values may be logged by the MKx for debug purposes.
```

5.1.2 Type definitions

5.1.2.1 tMKxC2XSecCmd

The tMKxC2XSecCmd data structure is the security command message format. It is used to send a security command, including the message payload, to the ECDSA accelerator. For full details see the NXP C2X Security API document [1], however for information purposes, the APDU command data structure is repeated here.
The class and instruction fields describe the command message type, specific to the ECDSA accelerator algorithm. Details of these fields are described in Section 5.1.3. The USN field is the unique sequence number, which is provides a unique ID which is returned with the security response message. The LE is the length of the corresponding response message.

```
/**
 * C2X security command message
 * +------+------+------+------+------+---...---+------+
 * | CLA  | INS  | USN0 | USN1 |  LC  | Payload |  LE  |
 * +------+------+------+------+------+---...---+------+
 *    1      1      1      1      1      'LC'      1
 */
typedef struct MKxC2XSecCmd
{
  /// CLA - Class Byte.
  uint8_t CLA;
  /// INS - Instruction Byte.
  uint8_t INS;
  /// USN0,USN1 - 2 Byte USN field.P1 and P2 fields are reused for USN
  uint8_t USN[2];
  /// LC - Length in Bytes of the payload. (0x01 ... 0xFF)
  uint8_t LC;
  /// Payload - Command Payload.
  tMKxC2XSecCmdPL Payload[0];
  /// LE - Expected Length of response. (0x01 ... 0xFF)
  uint8_t LE;
} __attribute__((__packed__)) tMKxC2XSecCmd;
```

5.1.2.2 tMKxC2XSecRsp

The tMKxC2XSecRsp data structure is used to return the result of the security operation. For full details see the NXP C2X Security API document [1], however for information purposes, the APDU response data structure is repeated here.
The result is contained in the Payload not the SW field. The USN field is the unique sequence number of the corresponding original command message.

```
/**
 * C2X security response message
 * +------+------+---...---+------+------+
 * | USN0 | USN1 | Payload |  SW1 |  SW2 |
```

```
 * +------+------+---...---+------+------+
 *    1      1       'LE'      1      1
 */
typedef struct MKxC2XSecRsp
{
  /// USN0, USN1
  uint8_t USN[2];
  /// Payload - Response Payload.
  tMKxC2XSecRspPL Payload[0];
  /// SW1,SW2 indicate the response code.
  /// Refer to C2X Security API document for valid error codes
  uint8_t SW[2];
} __attribute__((__packed__)) tMKxC2XSecRsp;
```

5.1.2.3 tMKxC2XSecAPDU

This is the application protocol data unit (APDU) data structure which is a union data structure of the security message command, security message response and a generic data field. Note payload aligned data structures tMKxC2XSecCmdPA and tMKxC2XSecRspPA are used to 64-bit align the command and response messages separately.

```
/// Wrapper to ensure tMKxC2XSecCmd.Payload is 64-bit aligned (payload aligned),
/// Data[0] and End[0] provide useful offsets into the structure without adding
/// to the overall size.
typedef struct MKxC2XSecCmdPA
{
  /// Padding bytes to ensure tMKxC2XSecCmd.Payload is 64-bit aligned
  uint8_t Padding[3];
  /// Command message data
  uint8_t Data[0];
  /// Message structure, now with aligned .Payload member
  tMKxC2XSecCmd Cmd;
  /// End of structure
  uint8_t End[0];
} __attribute__((__packed__)) tMKxC2XSecCmdPA;

/// Wrapper to ensure tMKxC2XSecRsp.Payload is 64-bit aligned (payload aligned),
/// Data[0] and End[0] provide useful offsets into the structure without adding
/// to the overall size.
typedef struct MKxC2XSecRspPA
{
  /// Padding bytes to ensure tMKxC2XSecRsp.Payload is 64-bit aligned
  uint8_t Padding[6];
  /// Raw response message data
  uint8_t Data[0];
  /// Message structure with aligned .Payload member
  tMKxC2XSecRsp Rsp;
  /// End of structure
```

```
  uint8_t End[0];
} __attribute__((__packed__)) tMKxC2XSecRspPA;

/**
 * C2X security message
 */
typedef union MKxC2XSecAPDU
{
  /// Command APDU (payload aligned)
  tMKxC2XSecCmdPA C;
  /// Response APDU (payload aligned)
  tMKxC2XSecRspPA R;
} __attribute__((__packed__)) tMKxC2XSecAPDU;
```

5.1.2.4 tMKxC2XSec

This is the top level security message data structure which includes the USB message header and the security command or response APDU.

```
/**
 * C2X security request/indication
 */
typedef struct MKxC2XSec
{
  /// Interface Message Header (reserved area for LLC usage)
  tMKxIFMsg Hdr;
  /// C2X Security API APDU
  tMKxC2XSecAPDU APDU;
} __attribute__((__packed__)) tMKxC2XSec;
```

5.1.3 Security Command Types

The supported enumerated values of the security class and instruction fields are detailed below. Note these enums are not declared in the LLC-API.h header file.

```
typedef enum MKxC2XSecClass
{
  /// Security class for the ECDSA accelerator
  SEC_CLASS_ECDSA_ACCELERATOR = 0xE0,
} eMKxC2XSecClass;

typedef enum MKxC2XSecInst
{
  /// Message instruction to reconstruct a public key
  SEC_INST_RECONSTRUCT_PUBLIC_KEY = 0x1A,
  /// Message instruction to decompress a public key
  SEC_INST_DECOMPRESS_PUBLIC_KEY = 0x6C,
```

```
  /// Message instruction to verify a message
  SEC_INST_VERIFY_MESSAGE = 0x6B,
} eMKxC2XSecInst;
```

5.1.4 Security Command Payloads

The payload portion (tMKxC2XSecCmdPL) of a command message (tMKxC2XSecCmd) is declared as a union of three possible message formats, corresponding to the three different command operations possible.

```
/// Security command payload structure
typedef union MKxC2XSecCmdPL
{
  /// Verify Signature of Hash command message
  tMKxC2XSecVSoH VerifySigOfHash;
  /// Decompress Public Key command message
  tMKxC2XSecDPK DecompEccPubKey;
  /// Reconstruct ECC Public Key command message
  tMKxC2XSecREPK ReconEccPubKey;
  /// Raw payload data
  uint8_t Data[0];
}__attribute__((__packed__)) tMKxC2XSecCmdPL;
```

The data structures for each message type are declared within the API and are listed below.

```
/// Verify Signature of Hash
/// See SAF5x00 Security user manual v0.5 (24 Jan 2017) Sec 4.1.1
typedef struct MKxC2XSecVSoH
{
  /// Public key of the entity that created the signature
  tMKxC2XSecPair PubKey;
  /// Hash protected by signature
  tMKxC2XSecHash E;
  /// The signature over the hash, to be verified (R)
  tMKxC2XSecSig R;
  /// The signature over the hash, to be verified (S)
  tMKxC2XSecSig S;
  /// Identifies ECC curve used to verify the ECC public key
  tMKxC2XSecCId CurveId;
}__attribute__((__packed__)) tMKxC2XSecVSoH;

/// Decompress Public Key
/// See SAF5x00 Security user manual v0.5 (24 Jan 2017) Sec 4.1.2
typedef struct MKxC2XSecDPK
{
  /// ECC Public key to decompressed
  tMKxC2XSecCompPubKey MKxC2XSecCompPubKey;
  /// Identifies ECC curve used to decompress ECC public key
```

```
  tMKxC2XSecCId CurveId;
}__attribute__((__packed__)) tMKxC2XSecDPK;

/// Reconstruct ECC Public Key
/// See SAF5x00 Security user manual v0.5 (24 Jan 2017) Sec 4.1.3
typedef struct MKxC2XSecREPK
{
  /// Hash value used in derivation of ECC public key
  tMKxC2XSecHash hvij;
  /// Public reconstruction value used in derivation of ECC public key
  tMKxC2XSecPair RVij;
  /// Public key of Pseudonym CA used in derivation of the ECC public key
  tMKxC2XSecPair Spca;
  /// Type of ECC curve used to reconstruct the ECC public key
  tMKxC2XSecCId CurveId;
}__attribute__((__packed__)) tMKxC2XSecREPK;
```

The message format use the following general types.

```
/// ECDSA Security Curve Identifiers
typedef enum MKxC2XSecCurveId
{
  MKXC2XSEC_CID_NIST256          = 0u, /// NIST curve param
  MKXC2XSEC_CID_BRAINPOOL_P256R1 = 1u, /// Brainpool curve P256r1 param
  MKXC2XSEC_CID_BRAINPOOL_P256T1 = 2u, /// Brainpool curve P256t1 param

  MKXC2XSEC_CID_COUNT
} eMKxC2XSecCurveId;
/// Public key curve id
typedef uint8_t tMKxC2XSecCId;

/// Public key signature
typedef uint8_t tMKxC2XSecSig[32];
/// Public key hash
typedef uint8_t tMKxC2XSecHash[32];
/// Public key coordinate
typedef uint8_t tMKxC2XSecCoord[32];

/// Public key pair
typedef struct MKxC2XSecPair
{
  /// X coordinate for elliptical signature
  tMKxC2XSecCoord X;
  /// Y coordinate for elliptical signature
  tMKxC2XSecCoord Y;
}__attribute__((__packed__)) tMKxC2XSecPair;

/// Compressed public key
```

```
typedef struct MKxC2XSecCompPubKey
{
  /// X coordinate for elliptical signature
  tMKxC2XSecCoord X;
  /// The least significant _bit_ of the Y coordinate
  uint8_t Ybit;
}__attribute__((__packed__)) tMKxC2XSecCompPubKey;
```

5.1.5 Security Response Payloads

The payload portion (tMKxC2XSecDspPL) of a security command response message (tMKxC2XSecRsp) is declared as a union of two possible message formats.

```
/// Security command response payload structure
typedef union MKxC2XSecRspPL
{
  /// Verify Signature of Hash command response
  tMKxC2XSecResRsp ResRsp;
  /// Decompress Public Key and Reconstruct Public Key commands response
  tMKxC2XSecPubKeyRsp PubKeyRsp;
  /// Raw payload data
  uint8_t Data[0];
}__attribute__((__packed__)) tMKxC2XSecRspPL;
```

The data structures for each message type are declared within the API and are listed below.

```
/// Result of Verify Signature of Hash command
/// See SAF5x00 Security user manual v0.5 (24 Jan 2017) Sec 4.1.1
typedef struct MKxC2XSecResRsp
{
  /// The result of the verification
  tMKxC2XSecVerRes VerResult;
}__attribute__((__packed__)) tMKxC2XSecResRsp;

/// Public Key response structure
/// See SAF5x00 Security user manual v0.5 (24 Jan 2017) Sec 4.1.2 and 4.1.3
typedef struct MKxC2XSecPubKeyRsp
{
  /// Public Key response
  tMKxC2XSecPair PubKey;
}__attribute__((__packed__)) tMKxC2XSecPubKeyRsp;
```

These data structures make use of the following sub-types.

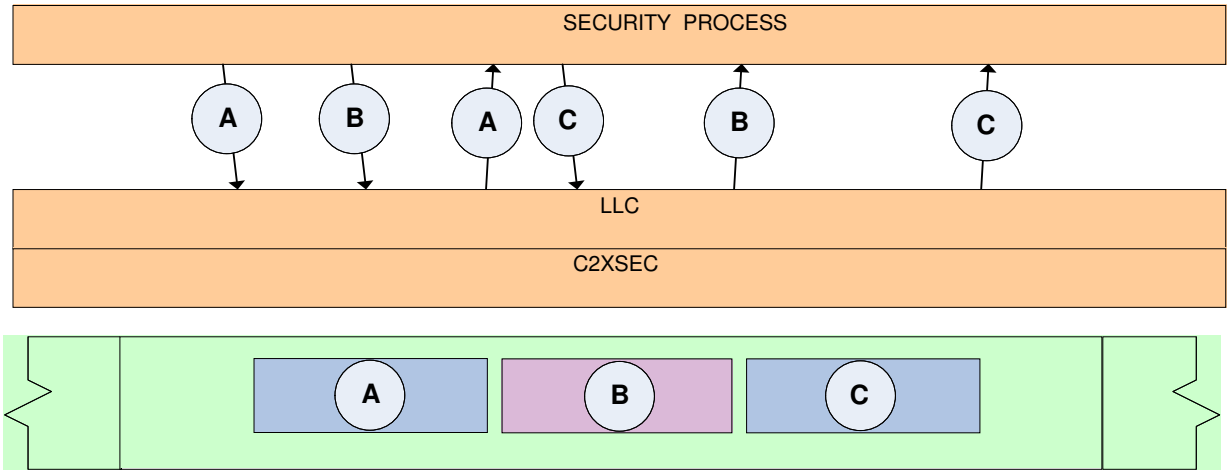
```
/// Security verification results
typedef enum MKxC2XSecVerifyResult
{
  MKXC2XSEC_VERIFY_SUCCESS = 0u,
  MKXC2XSEC_VERIFY_FAILURE = 1u
} eMKxC2XSecVerifyResult;
```

```
/// Security verification result
typedef uint8_t tMKxC2XSecVerRes;

/// Public key pair
typedef struct MKxC2XSecPair
{
  /// X coordinate for elliptical signature
  tMKxC2XSecCoord X;
  /// Y coordinate for elliptical signature
  tMKxC2XSecCoord Y;
}__attribute__((__packed__)) tMKxC2XSecPair;
```

5.2 Usage Example

5.2.1 Message Verification



The sequence below shows an example of the flow of messaging between the layers when performing message verification.

1. The stack assigns a callback function to process responses by mapping it to the fC2XSec_ResponseInd().
2. The stack submits a message (A) for verification using fC2XSec_CommandReq()
3. The LLC passes the message (A) to the SAF5x00 for processing
4. The stack submits a message (B) for verification using fC2XSec_CommandReq()
5. The LLC passes the message (B) to the SAF5x00 for processing
6. The SAF5x00 returns the verification result of message (A) to the LLC
7. The LLC notifies the stack by calling the fC2XSec_ResponseInd() callback function, with the response for message (A).
8. The stack submits a message (C) for verification using fC2XSec_CommandReq()
9. The LLC passes the message (C) to the SAF5x00 for processing
10. The SAF5x00 returns the verification result of message (B) to the LLC
11. The LLC notifies the stack by calling the fC2XSec_ResponseInd() callback function, with the response for message (B).
12. The SAF5x00 returns the verification result of message (C) to the LLC
13. The LLC notifies the stack by calling the fC2XSec_ResponseInd() callback function, with the response for message (C).

6 References

[1] C2X Security API, Rev 0.2 – July 10 2014, NXP Semiconductors