Jennic
TECHNOLOGY FOR A CHANGING WORLD

802.15.4 Stack API

Reference Manual

JN-RM-2002
Revision 1.7
10-Jan-2007

Disclaimer

The contents of this document are subject to change without notice. Customers are advised to consult with JENNIC commercial representatives before ordering.

The information and circuit diagrams in this document are presented as examples of semiconductor device applications, and are not intended for incorporation in devices for actual use. In addition, JENNIC is unable to assume responsibility for infringement of any patent rights or other rights of third parties arising from the use of this information or circuit diagrams.

No license is granted by its implication or otherwise under any patent or patent rights of JENNIC Ltd

"Typical" parameters, which are provided in this document, may vary in different applications and performance may vary over time. All operating parameters must be validated for each customer application by the customer's own technical experts.

CAUTION:
Customers considering the use of our products in special applications where failure or abnormal operation may directly affect human lives or cause physical injury or property damage, or where extremely high levels of reliability are demanded (such as aerospace systems, atomic energy controls, vehicle operating controls, medical devices for life support, etc.) are requested to consult with JENNIC representatives before such use. JENNIC customers using or selling products incorporating JENNIC IP for use in such applications do so at their own risk and agree to fully indemnify JENNIC for any damages resulting from such improper use or sale.

Contents

Disclaimer 2

Contents 3

About this Manual 5
Organisation 5
Conventions 5
Acronyms and Abbreviations 5
Revision History 6

1 Introduction 7

2 Service Access Point API 8
2.1 Background information 8
2.1.1 Service Access Point fundamentals 8
2.1.2 Blocking and non-blocking operation 8
2.1.3 Call/callback interface 9
2.2 Implementation 10
2.3 API functions 11
2.3.1 Send Request/Response 11
2.3.2 Register Deferred Confirm/Indication callbacks 11

3 MAC / Network layer interface 12
3.1 Network layer to MAC layer interface 12
3.1.1 NWK to MLME 13
3.1.2 NWK to MCPS 17
3.2 MAC layer to Network layer interface 20
3.2.1 MLME/MCPS to NWK 20
3.2.2 MAC Settings 23

4 IEEE 802.15.4 MAC/PHY features 24
4.1 Status returns 24
4.2 PAN Information Base 25
4.2.1 MAC Layer PIB access 25
4.2.2 Physical Layer PIB access 28
4.3 MAC Reset 31
4.3.1 Reset Example 31
4.4 Scan 32
4.4.1 Energy Detect Scan 32
4.4.2 Active Scan 32
4.4.3 Passive Scan 32
4.4.4 Orphan Scan 32
4.4.5 Scan Request 33
4.4.6 Scan Confirm 33
4.4.7 Orphan Indication 36
4.4.8 Orphan Response 37
4.4.9 Comm Status Indication 37
4.4.10 Examples 39

4.5 Start 41
4.5.1 Start request 41
4.5.2 Start confirm 42
4.5.3 Examples 43
4.6 Synchronisation 44
4.6.1 Sync request 44
4.6.2 Sync loss indication 45
4.6.3 Beacon Notify Indication 46
4.6.4 Poll Request 47
4.6.5 Poll Confirm 47
4.6.6 Examples 48
4.7 Association 49
4.7.1 Associate Request 50
4.7.2 Associate Confirm 51
4.7.3 Associate Indication 52
4.7.4 Associate Response 52
4.7.5 Comm Status Indication 53
4.7.6 Examples 54
4.8 Disassociation 57
4.8.1 Disassociate Request 57
4.8.2 Disassociate Confirm 58
4.8.3 Disassociate Indication 59
4.8.4 Examples 59
4.9 Data transmission and reception 60
4.9.1 Data Request 60
4.9.2 Data Confirm 61
4.9.3 Data Indication 62
4.9.4 Purge Request 63
4.9.5 Purge Confirm 63
4.9.6 Examples 64
4.10 Rx Enable 66
4.10.1 Rx Enable Request 67
4.10.2 Rx Enable Confirm 67
4.10.3 Examples 68
4.11 Guaranteed Time Slots (GTS) 68
4.11.1 GTS Request 69
4.11.2 GTS Confirm 69
4.11.3 GTS Indication 70
4.11.4 Examples 71

Appendix A 73

References 74

About this Manual

This manual provides a detailed reference for the Jennic 802.15.4 Stack API (Application Programming Interface). This facilitates control of the 802.15.4 MAC hardware within the Jennic JN5121 and JN513x single-chip wireless microcontrollers. This software is supplied as a set of precompiled library builds with the Jennic Software Developer's Kit (SDK).

Note: This manual was previously called the 802.15.4 MAC Software Reference Manual.

Organisation

This manual consists of four chapters and an appendix:

- Chapter 1 defines the scope of the manual.
- Chapter 2 describes the implementation of the API.
- Chapter 3 describes in detail the MAC-Network Layer Interface.
- Chapter 4 outlines the MAC/PHY features.
- Appendix A describes how to identify modules and devices.

Conventions

Code fragments, function prototypes or filenames are represented by `Courier` typeface. When referring to constants or functions defined in the code they are emboldened **`like so.`**

Acronyms and Abbreviations

ACL	Access Control List
AHI	Application Hardware Interface
API	Application Programming Interface
CAP	Contention Access Period
FFD	Full Function Device
GTS	Guaranteed Time Slot
MAC	Medium Access Control
MCPS	MAC common Part Sublayer
MLME	MAC sub-Layer Management Entity
NWK	Network (layer)
PAN	Personal Area Network
PHY	Physical
PIB	PAN Information Base
RFD	Reduced Function Device
SAP	Service Access Point

Revision History

Version	Date	Description
1.0	11-Sept-2005	First release
1.1	14-Nov-2005	Updated document style
1.2	27-Jan-2006	Aligned document with updated API
1.3	14-Mar-2006	Updated PIB access section
1.4	12-Apr-2006	Added Appendix A
1.5	06-Oct-2006	Name of API changed from 802.15.4 MAC Software to 802.15.4 Stack API
1.6	16-Oct-2006	Updated PHY PIB access description
1.7	10-Jan-2007	Updated for JN513x chip series

1 Introduction

This document describes the structure of the 802.15.4 Stack API to be used in conjunction with the Jennic JN5121 and JN513x single-chip wireless microcontrollers. The interface described is exposed at the level of transactions into and out of the stack; this allows different types of interfaces to be written which deal with buffering messages in ways best suited to the type of application that uses the stack. An example of this is found in the description of the Demonstration Application, which takes the API described here, and puts a queue-based interface on top for storing and dealing with information entering and leaving the stack.

Note: This API was previously known as the 802.15.4 MAC Software.

2 Service Access Point API

2.1 Background information

This section gives some information on the background behind how the API has been implemented.

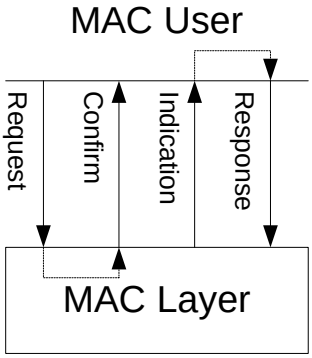
2.1.1 Service Access Point fundamentals

[2] specifies the service primitives, which pass between what is described as the “N-user” and the “N-layer”. “N” is simply an abstract term to describe a specific layer of the protocol stack under consideration.

This document considers the service primitives, which pass between the user of the 802.15.4 MAC User and the 802.15.4 MAC Layer as specified in [1].

In general, the service primitives are classified as follows:

- Request
- Confirm
- Indication
- Response



A Request transaction is initiated by the MAC User and may solicit a Confirm. An Indication transaction is initiated by the MAC Layer and may solicit a Response.

As this is purely a reference model, a specific implementation needs to be built upon the reference model. This section describes the implementation based on the reference model.

2.1.2 Blocking and non-blocking operation

An implementation issue, which needs to be considered, is whether transactions are:

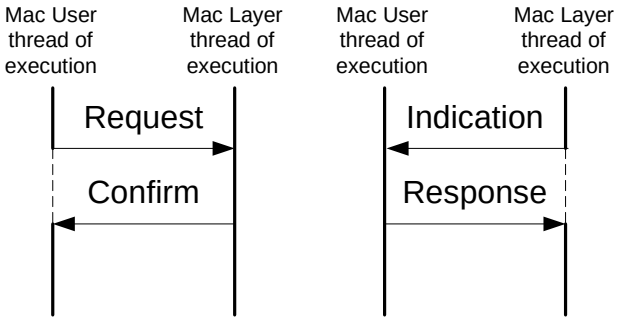
- Blocking (synchronous)
- Non-blocking (asynchronous).

2.1.2.1 Blocking transaction

A blocking, or synchronous transaction occurs when the initiator of the transaction explicitly waits for information coming back from the target of the transaction.

In the case of a Request, the MAC User would wait for a Confirm before carrying on processing.

In the case of an Indication, the MAC Layer would wait for a Response before carrying on processing.

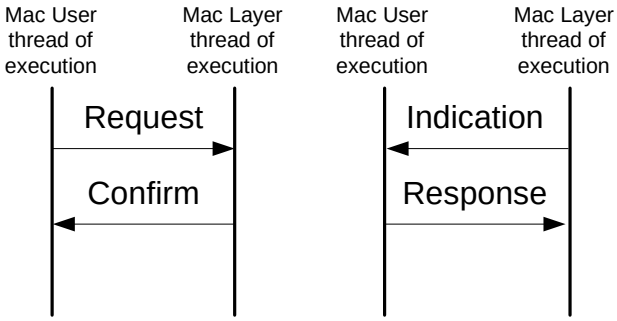


2.1.2.2 Non-blocking transaction

A non-blocking transaction occurs when the initiator of the transaction does not explicitly wait for information to come back from the target of the transaction before continuing its own execution

In the case of a Request, the Application would send the Request then carry on processing; the Confirm would come back some time later (i.e. asynchronously) and be processed accordingly.

In the case of an Indication, the MAC Layer would send the Indication then carry on processing; the Response would come back asynchronously and be processed accordingly.

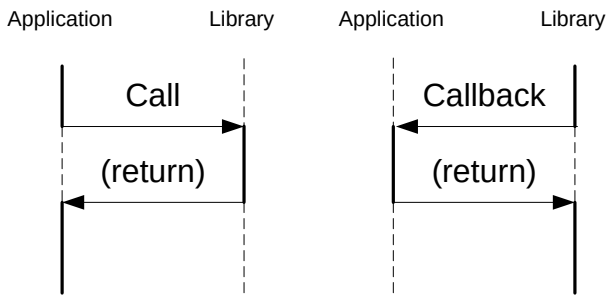


2.1.3 Call/callback interface

The most straightforward API is via a function call/callback interface.

A function call is made from the application to the library in the applications thread of execution. The function can be called directly by the application

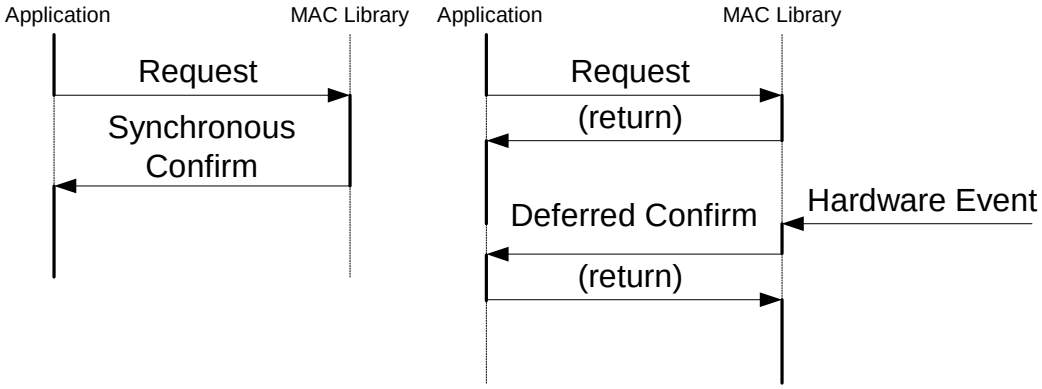
A function callback is made from a library to the application in the library's thread of execution. The callback function is registered with the library by the application, and is available for the library to call.



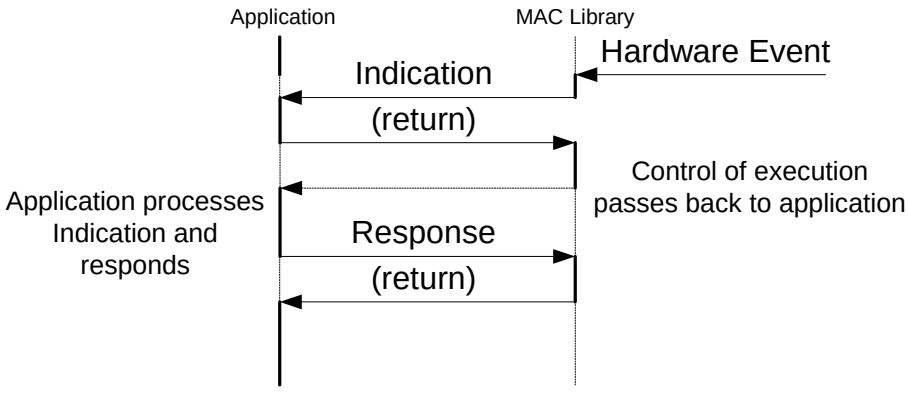
Note that the two threads of execution do not necessarily have to be the same, but the key point is that the call/callback is executed in the thread of execution of the caller of the function.

2.2 Implementation

Based on the above, an implementation can be formed which satisfies all cases of blocking and non-blocking operations for the primitives based on a call/callback interface. This is illustrated as follows:



Request/Confirm processing



Indication/Response processing

The one restriction is that there is no synchronous response to an indication. This is not really a problem as:

- Most indications do not solicit a response as they represent an event
- Control of processing is governed by the higher layers and thus the response may need to be formed in a different thread of execution.
- The MAC layer is implemented as a finite state machine and is thus implicitly able to handle asynchronous transactions.

Conversely, it is useful to have a synchronous Confirm to many Requests, such as PIB Get and Set, which can be satisfied by a synchronous transaction; also, if a Request results in an error, this may often be returned straight away.

2.3 API functions

So, from the above diagram, the call/callback interface is implemented via two interface functions:

- Send Request/Response
- Register Deferred Confirm/Indication callbacks

2.3.1 Send Request/Response

The Request/Response functions are used by the Application to send a Request or a Response to the appropriate SAP.

2.3.2 Register Deferred Confirm/Indication callbacks

The callback registration function is used by the Application to register two Callbacks for Deferred Confirms and Indications. A two-phase callback system is used as it gives the Application control of the buffer allocation. This allows the Application to easily implement a basic queuing system for Deferred Confirms and Indications, which are always handled asynchronously.

The two callbacks are:

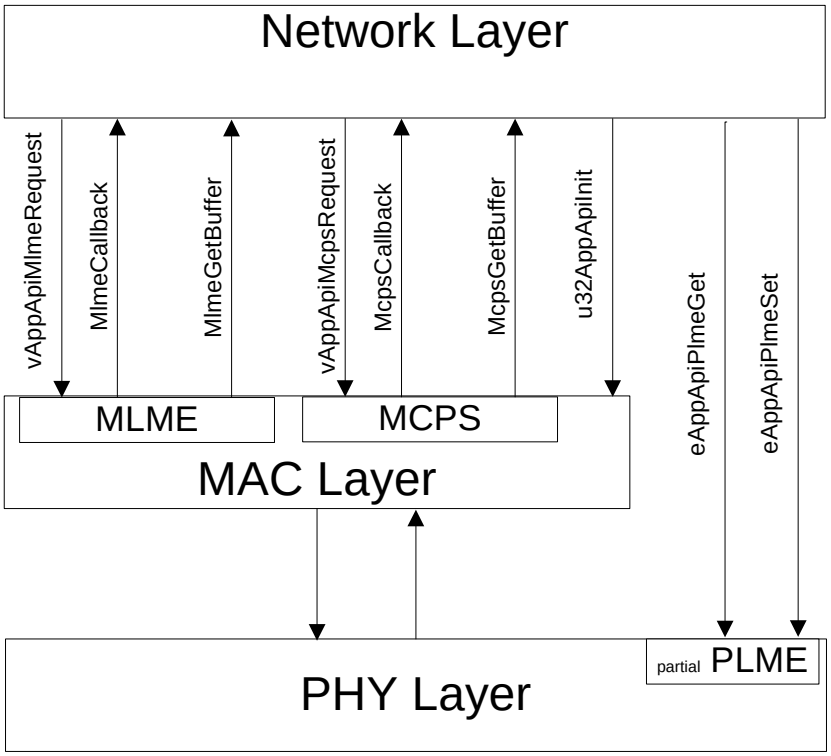
- Get Buffer
- Post

The GetBuffer callback is called by the MAC whenever it needs to get a buffer into which it will write the Deferred Confirmation or the Indication.

The Post callback is called by the MAC at the point it wishes to post the Deferred Confirm or Indication to the Application

3 MAC / Network layer interface

This section describes the interface between the Network layer and the MAC layer. The figure provides an overview of the functions making up this interface.



The following sections first describe the functions allowing requests from the Network layer to the MAC layer and then the (callback) functions allowing the MAC layer to request the Network layer to allocate buffer space and to pass information back to the Network layer.

3.1 Network layer to MAC layer interface

The NWK to MLME and NWK to MCPS interfaces are implemented as calls from the NWK layer to routines provided by the MAC. The general procedure to use these calls is to fill in a structure representing a request to the MAC and either receive a synchronous confirm, for which space must be allocated, or to expect a deferred (asynchronous) confirm at some time later. The application may elect to perform other tasks while waiting for a deferred confirm, or if there is nothing for it to do, go to sleep to save power.

3.1.1 NWK to MLME

Declaration

```
PUBLIC void
vAppApiMlmeRequest(MAC_MlmeReqRsp_s *psMlmeReqRsp,
                   MAC_MlmeSyncCfm_s *psMlmeSyncCfm);
```

Inputs

`MAC_MlmeReqRsp_s *psMlmeReqRsp`	Pointer to a structure holding the request to the MLME interface
`MAC_MlmeSyncCfm_s *psMlmeSyncCfm`	Pointer to a structure used to hold the result of a synchronous confirm to a request over the MLME interface

Outputs

None

Description

Routine used to pass MLME requests from the NWK layer or Application to the MAC.

The psMlmeReqRsp parameter is a pointer to a structure holding the request to the MLME. The structure is of type `MAC_MlmeReqRsp_s`, defined below

```
/**
 * MLME Request/Response
 *
 * The object passed to vAppApiMlmeRequest containing
 * the request
 */
      typedef struct
      {
          uint8                 u8Type;
          uint8                 u8ParamLength;
          uint16                u16Pad;
          MAC_MlmeReqRspParam_u uParam;
      } MAC_MlmeReqRsp_s;
```

The structure consists of 4 fields. The first, `u8Type` defines the type of request or response that the structure carries; values carried in this field are defined in the enumeration `MAC_MlmeReqType_e` shown below:

```
      /* Enumeration of MAC MLME Request/Response
       * Must not exceed 256 entries
       */
      typedef enum
      {
          MAC_MLME_REQ_ASSOCIATE = 0,
          MAC_MLME_REQ_DISASSOCIATE,
          MAC_MLME_REQ_GET,
          MAC_MLME_REQ_GTS,
          MAC_MLME_REQ_RESET,
          MAC_MLME_REQ_RX_ENABLE,
          MAC_MLME_REQ_SCAN,
          MAC_MLME_REQ_SET,
          MAC_MLME_REQ_START,
          MAC_MLME_REQ_SYNC,
          MAC_MLME_REQ_POLL,

          MAC_MLME_RSP_ASSOCIATE,
          MAC_MLME_RSP_ORPHAN,
```

```
    MAC_MLME_REQ_VS_EXTADDR,
    NUM_MAC_MLME_REQ           /* (endstop) */
} MAC_MlmeReqRspType_e;
```

The second field u8ParamLength carries the size in bytes of the parameter associated with the request. The parameter has a maximum size of 255 bytes

The third field ensures proper alignment of the fourth field.

The fourth field uParam is of type `MAC_MlmeReqRspParam_u`, a union of all the data structures associated with the requests listed in `MAC_MlmeReqRspType_e`

The union is defined as follows:

```
/* MLME Request/Response Parameter union
 * Union of all the possible MLME Requests and Responses,
 * also including the vendor-specific requests
 */
union
{
    /* MLME Requests */
    MAC_MlmeReqAssociate_s    sReqAssociate;
    MAC_MlmeReqDisassociate_s sReqDisassociate;
    MAC_MlmeReqGet_s          sReqGet;
    MAC_MlmeReqGts_s          sReqGts;
    MAC_MlmeReqReset_s        sReqReset;
    MAC_MlmeReqRxEnable_s     sReqRxEnable;
    MAC_MlmeReqScan_s         sReqScan;
    MAC_MlmeReqSet_s          sReqSet;
    MAC_MlmeReqStart_s        sReqStart;
    MAC_MlmeReqSync_s         sReqSync;
    MAC_MlmeReqPoll_s         sReqPoll;

    /* MLME Responses */
    MAC_MlmeRspAssociate_s    sRspAssociate;
    MAC_MlmeRspOrphan_s       sRspOrphan;

    /* Vendor Specific Requests */
    MAC_MlmeReqVsExtAddr_s    sReqVsExtAddr;


} MAC_MlmeReqRspParam_u;
```

The individual data structures that make up the union will be dealt with in more detail in the section on MAC and PHY features, which explains the operations that the higher layer can request using this interface.

The psMlmeSyncCfm parameter is a pointer to a structure holding the results of the MLME request (the confirm), generated if the request executes synchronously (ie returns with the results immediately, rather than the results being posted as a deferred confirm some time later). If the MLME request is one that generates a deferred confirm, a synchronous confirm is still generated but with a status of MAC_MLME_CFM_DEFERRED (see below).

The structure is of type MAC_MlmeSyncCfm_s defined below

```
/* MLME Synchronous Confirm
 *
 * The object returned by vAppApiMlmeRequest containing
 * the synchronous confirm
 * All Confirms may also be sent asynchronously via the
 * registered Deferred Confirm/Indication callback.
 * This is notified by returning MAC_MLME_CFM_DEFERRED.
 * The confirm type is implied, corresponding to the
 * request
 */
typedef struct
{
    uint8                  u8Status;
    uint8                  u8ParamLength;
    uint16                 u16Pad;
    MAC_MlmeSyncCfmParam_u uParam;
} MAC_MlmeSyncCfm_s;
```

The first field carries the status of the request which caused the confirm, and the values it may take are defined by the enumeration MAC_MlmeSyncCfmStatus_e

```
/* Synchronous confirm status
 *
 * Indicates in the synchronous confirm whether:
 * (1) The request was processed without error
 * (2) The request was processed with errors
 * (3) The confirm will be deferred and posted via the
 *     Deferred Confirm/Indication callback
 * (4) It is a dummy confirm to a Response.
 * Note: must not exceed 256 entries
 */
typedef enum
{
    MAC_MLME_CFM_OK,
    MAC_MLME_CFM_ERROR,
    MAC_MLME_CFM_DEFERRED,
    MAC_MLME_CFM_NOT_APPLICABLE,
    NUM_MAC_MLME_CFM                /* (endstop) */
} MAC_MlmeSyncCfmStatus_e;
```

The second field u8ParamLength carries the size in bytes of the parameter associated with the confirm. The parameter has a maximum size of 255 bytes

The third field ensures proper alignment of the fourth field.

The fourth field uParam is of type MAC_MlmeSyncCfmParam_u, a union of all the data structures associated with the confirms that can come back from requests to the MLME, including vendor-specific requests

The union is defined as follows:

```
/* MLME Synchronous Confirm Parameter union
 *
 * Union of all the possible MLME Synchronous Confirms,
 * including the vendor-specific confirms
 */
typedef union
{
    MAC_MlmeCfmAssociate_s    sCfmAssociate;
    MAC_MlmeCfmDisassociate_s sCfmDisassociate;
    MAC_MlmeCfmGet_s          sCfmGet;
    MAC_MlmeCfmGts_s          sCfmGts;
    MAC_MlmeCfmScan_s         sCfmScan;
    MAC_MlmeCfmSet_s          sCfmSet;
    MAC_MlmeCfmStart_s        sCfmStart;
    MAC_MlmeCfmPoll_s         sCfmPoll;
    MAC_MlmeCfmReset_s        sCfmReset;
    MAC_MlmeCfmRxEnable_s     sCfmRxEnable;
    MAC_MlmeCfmVsBbcReg_s     sCfmVsBbcReg;
    MAC_MlmeCfmVsRdReg_s      sCfmVsRdReg;
} MAC_MlmeSyncCfmParam_u;
```

Examples of using the call and setting up the parameters and interpreting the results will be given throughout the remainder of the document.

3.1.2 NWK to MCPS

Declaration

```
PUBLIC void
vAppApiMcpsRequest(MAC_McpsReqRsp_s *psMcpsReqRsp,
                   MAC_McpsSyncCfm_s *psMcpsSyncCfm);
```

Inputs

`MAC_McpsReqRsp_s *psMcpsReqRsp`	Pointer to a structure holding the request to the MCPS interface
`MAC_McpsSyncCfm_s *psMcpsSyncCfm`	Pointer to a structure used to hold the result of a synchronous confirm to a request over the MCPS interface

Outputs

None

Description

Routine used to pass MCPS requests from the NWK layer or Application to the MAC.

The psMcpsReqRsp parameter is a pointer to a structure holding the request to the MCPS. The structure is of type `MAC_MlmeReqRsp_s`, defined below

```
/* MCPS Request/Response object
 *
 * The object passed to vAppApiMcpsRequest containing the
 * request/response
 */
typedef struct
{
    uint8                u8Type;
    uint8                u8ParamLength;
    uint16               u16Pad;
    MAC_McpsReqRspParam_u uParam;
} MAC_McpsReqRsp_s;
```

The structure consists of 4 fields. The first, `u8Type` defines the type of request or response that the structure carries; values carried in this field are defined in the enumeration `MAC_McpsReqRspType_e` shown below:

```
/* MAC MCPS Request/Response enumeration.
 * Note must not exceed 256 entries
 */
typedef enum
{
    MAC_MCPS_REQ_DATA = 0,
    MAC_MCPS_REQ_PURGE,
    NUM_MAC_MCPS_REQ        /* (endstop) */
} MAC_McpsReqRspType_e;
```

The second field `u8ParamLength` carries the size in bytes of the parameter associated with the request. The parameter has a maximum size of 255 bytes

The third field ensures proper alignment of the fourth field.

The fourth field `uParam` is of type `MAC_McpsReqRspParam_u`, a union of all the data structures associated with the requests listed in `MAC_McpsReqRspType_e`

The union is defined as follows:

```
/* MCPS Request/Response Parameter union
```

```
 * Note there are no Responses currently specified
 */
typedef union
{
    MAC_McpsReqData_s  sReqData;   /* Data request */
    MAC_McpsReqPurge_s sReqPurge;  /* Purge request */
} MAC_McpsReqRspParam_u;
```

The individual data structures, which make up the union, will be dealt with in more detail in the section on MAC and PHY features, which explains the operations that the NWK layer or Application can request using this interface.

The psMcpsSyncCfm parameter is a pointer to a structure holding the results of the MCPS request (the confirm), generated if the request executes synchronously (i.e. returns with the results immediately, rather than the results being posted as a deferred confirm some time later).

The structure is of type `MAC_McpsSyncCfm_s defined below`

```
/**
 * MCPS Synchronous Confirm
 * The object returned by vAppApiMcpsRequest containing
 * the synchronous confirm.
 * The confirm type is implied as corresponding to the
 * request
 * All Confirms may also be sent asynchronously via the
 * registered Deferred Confirm/Indication callback;
 * this is notified by returning MAC_MCPS_CFM_DEFERRED.
 */
typedef struct
{
    uint8                  u8Status;
    uint8                  u8ParamLength;
    uint16                 u16Pad;
    MAC_McpsSyncCfmParam_u uParam;
} MAC_McpsSyncCfm_s;
```

The first field carries the status of the request which caused the confirm, and the values it may take are defined by the enumeration `MAC_McpsSyncCfmStatus_e`

```
/* Synchronous confirm status
 *
 * Indicates in the synchronous confirm whether:
 * (1) The request was processed without error
 * (2) The request was processed with errors
 * (3) The confirm will be deferred and posted via the
 *     Deferred Confirm/Indication callback
 * Note: must not exceed 256 entries
 */
typedef enum
{
    MAC_MCPS_CFM_OK,
    MAC_MCPS_CFM_ERROR,
    MAC_MCPS_CFM_DEFERRED,
    NUM_MAC_MCPS_CFM        /* (endstop) */
} MAC_McpsSyncCfmStatus_e;
```

The second field `u8ParamLength` carries the size in bytes of the parameter associated with the confirm. The parameter has a maximum size of 255 bytes.

The third field ensures proper alignment of the fourth field.

The fourth field `uParam` is of type `MAC_McpsSyncCfmParam_u`, a union of all the data structures associated with the confirms that can come back from requests to the MCPS, including vendor-specific requests

The union is defined as follows:

```
/* MCPS Synchronous Confirm Parameter union
 *
 * Union of all the possible MCPS Synchronous Confirms
 */
typedef union
{
    MAC_McpsCfmData_s  sCfmData;
    MAC_McpsCfmPurge_s sCfmPurge;
} MAC_McpsSyncCfmParam_u;
```

Examples of using the call and setting up the parameters and interpreting the results will be given throughout the remainder of the document.

3.2 MAC layer to Network layer interface

Communication from the MAC up to the application or network layer is through callback routines implemented by the upper layer and registered with the MAC at system initialisation. In this way, the upper layer can implement the method of dealing with indications and confirmations that suits it best.

3.2.1 MLME/MCPS to NWK

Declaration

```
PUBLIC uint32
u32AppApiInit(PR_GET_BUFFER prMlmeGetBuffer,
              PR_POST_CALLBACK prMlmeCallback,
              void *pvMlmeParam,
              PR_GET_BUFFER prMcpsGetBuffer,
              PR_POST_CALLBACK prMcpsCallback,
              void *pvMcpsParam);
```

Inputs

`PR_GET_BUFFER prMlmeGetBuffer`	Pointer to routine which is called by the MAC to provide a buffer to place the result of a deferred MLME callback or indication for sending to the network layer
`PR_POST_CALLBACK prMlmeCallback`	Pointer to routine which is called by the MAC to post (send) the buffer provided by the registered prMlmeGetBuffer routine up to the network layer
`void *pvMlmeParam`	Untyped pointer which is passed when calling the registered prMlmeGetBuffer and prMlmeCallback routines
`PR_GET_BUFFER prMcpsGetBuffer`	Pointer to routine which is called by the MAC to provide a buffer to place the result of a deferred MCPS callback or indication for sending to the network layer
`PR_POST_CALLBACK prMcpsCallback`	Pointer to routine which is called by the MAC to post (send) the buffer provided by the registered prMcpsGetBuffer routine up to the network layer
`void *pvMcpsParam`	Untyped pointer which is passed when calling the registered prMcpsGetBuffer and prMcpsCallback routines

Outputs

uint32 0 if initialisation failed, otherwise a 32-bit version number (most significant 16 bits are main revision, least significant 16 bits are minor revision)

Description

This routine registers five functions provided by the network layer, which are used by the MAC and the Integrated Peripherals API to communicate with the network layer.

Parameter 1: prMlmeGetBuffer

This is a routine that must provide a pointer to a buffer of type `MAC_DcfmIndHdr_s`, which can be used by the MAC to send the results of deferred (asynchronous) confirms as the result of a previous MLME Requests. The same routine will also be called by the MAC to provide space to send information to the network layer in the form of MLME Indications triggered by hardware events.

The network layer must provide a routine with the prototype

```
MAC_DcfmIndHdr_s *psMlmeDcfmIndGetBuf(void *pvParam)
```

which implements some form of buffer management which can return a pointer to a buffer of type `MAC_DcfmIndHdr_s`. At its simplest, this could be to return the address of a variable of this type known by the network layer, e.g.

```
PRIVATE MAC_DcfmIndHdr_s sAppBuffer;

PRIVATE MAC_DcfmIndHdr_s *
psMlmeDcfmIndGetBuf(void *pvParam)
{
    /* Return a handle to a MLME buffer */
    return &sAppBuffer;
}
```

although this implementation would be very limited in the number of responses or indications that could be handled at any time. Other suitable implementations within the network layer might be a queue, where the next free space is returned, or a pool of buffers which are allocated and freed by the network layer. In all cases it is up to the network layer to manage the freeing of buffers carrying deferred confirms and indications. If the network layer cannot provide a buffer it should return NULL, and the confirm/indication will be lost.

The pvParam parameter is provided as a pointer which can be used to carry further information between the MAC and network layer or vice versa when performing an MLME Get or Post, and contains `pvMlmeParam`, the third parameter to `u32AppApiInit`. This can be used for any purpose by the network layer and has no meaning to the MAC.

Parameter 2: prMlmeCallback

This routine is used to send the buffer provided by the above function to the network layer after the results of the MLME confirm or indication have been filled in. The network layer must provide a routine with the prototype

```
PRIVATE void
vMlmeDcfmIndPost(void *pvParam,
                 MAC_DcfmIndHdr_s *psDcfmIndHdr)
```

The routine expects always to successfully send the buffer it received from the network layer, which is not unreasonable, since the network layer is in charge of allocating the buffer in the first place. If the implementation is done in such a way that this might not be the case, the Send routine will have no way of signalling that it could not send the buffer up to the network layer. It is the responsibility of the network layer to provide sufficient buffers to be allocated to avoid losing confirms or indications

The pvParam parameter is provided as a pointer which can be used to carry further information between the MAC and network layer or vice versa when performing an MLME Get or Post, and contains `pvMlmeParam`, the third parameter to `u32AppApiInit`. This can be used for any purpose by the network layer and has no meaning to the MAC.

The psDcfmIndHdr parameter is a pointer to the buffer allocated in the `prMlmeGetBuffer` call carrying the information from the confirm/indication from the MAC to the network layer.

As an example of what a Post routine might do, consider the following

```
PRIVATE void
vMlmeDcfmIndPost(void *pvParam,
                 MAC_DcfmIndHdr_s *psDcfmIndHdr)
{
    /* Place incoming buffer on network layer input queue */
    vAddToQueue(psDcfmIndHdr);

    /* Signal the network layer that there is at least one
     * buffer to process.  If using a RTOS, this could be
     * a signal to the network layer to begin running to
     * process the buffer.  In a simple application a
     * variable might be polled as here
     */
    boNotEmpty = TRUE;
}
```

In the example, the interface between the MAC and network layer is a queue with enough entries to contain all the buffer pointers from a buffer pool managed by the network layer for the MLME confirm/indications. The Post routine places the buffer pointer on the queue and then signals the network layer that there is something there to process. This is all happening in the MAC thread of execution, which for a simple system will be in the interrupt context. At some stage the MAC thread will stop running and the network layer thread will continue; in this case it regularly polls the input queue and processes any entries it finds, before returning the buffer back to the buffer pool.

Parameter 3: pvMlmeParam

This is the value passed in calls to the above MLME routines.

Parameter 4: prMcpsGetBuffer

This routine has the same functionality as prMlmeGetBuffer, but is used to obtain a buffer for use with MCPS, rather than MLME, deferred confirmations or indications. The parameter passed with the call is pvMcpsParam.

Parameter 5: prMcpsCallback

This routine has the same functionality as prMlmeCallback, but is used to post a buffer containing a MCPS, rather than MLME, deferred confirmation or indication. The parameter passed with the call is pvMcpsParam.

Parameter 6: pvMcpsParam

This is the value passed in calls to the above MCPS routines.

3.2.2 MAC Settings

Declaration `PUBLIC void vAppApiSaveMacSettings(void);`

Inputs None

Outputs None

Description This function is used to enable the MAC to save settings in RAM before entering sleep mode with memory held up.

Declaration `PUBLIC void vAppApiRestoreMacSettings(void);`

Inputs None

Outputs None

Description This function is used when the device wakes up - it restores the MAC to the state that it was in before the device entered sleep mode.

Currently, this feature is only suitable for use in networks that do not use regular beacons, as it does not include a facility to resynchronise.

4 IEEE 802.15.4 MAC/PHY features

This section describes the features of the MAC and PHY in detail, showing the types of operations they support and the methods that the network layer can use to get access to them via the Request/Confirm and Indication/Response messages.

4.1 Status returns

In all the calls there are status values returned to indicate success or failure of the operation, defined by an enumeration **`MAC_enum_e`**, names and values shown in the table below.

This enumeration is defined in Table 64 (section 7.1.17) of the 802.15.4 specification (d18). Refer to the specification for definitive definitions.

Name	Value	Description
MAC_ENUM_SUCCESS	0x00	Success
MAC_ENUM_BEACON_LOSS	0xE0	Beacon loss after synchronisation request
MAC_ENUM_CHANNEL_ACCESS_FAILURE	0xE1	CSMA/CA channel access failure
MAC_ENUM_DENIED	0xE2	GTS request denied
MAC_ENUM_DISABLE_TRX_FAILURE	0xE3	Could not disable transmit or receive
MAC_ENUM_FAILED_SECURITY_CHECK	0xE4	Incoming frame failed security check
MAC_ENUM_FRAME_TOO_LONG	0xE5	Frame too long after security processing to be sent
MAC_ENUM_INVALID_GTS	0xE6	GTS transmission failed
MAC_ENUM_INVALID_HANDLE	0xE7	Purge request failed to find entry in queue
MAC_ENUM_INVALID_PARAMETER	0xE8	Out-of-range parameter in primitive
MAC_ENUM_NO_ACK	0xE9	No acknowledgement received when expected
MAC_ENUM_NO_BEACON	0xEA	Scan failed to find any beacons
MAC_ENUM_NO_DATA	0xEB	No response data after a data request
MAC_ENUM_NO_SHORT_ADDRESS	0xEC	No allocated short address for operation
MAC_ENUM_OUT_OF_CAP	0xED	Receiver enable request could not be executed as CAP finished
MAC_ENUM_PAN_ID_CONFLICT	0xEE	PAN ID conflict has been detected
MAC_ENUM_REALIGNMENT	0xEF	Coordinator realignment has been received
MAC_ENUM_TRANSACTION_EXPIRED	0xF0	Pending transaction has expired and data discarded
MAC_ENUM_TRANSACTION_OVERFLOW	0xF1	No capacity to store transaction
MAC_ENUM_TX_ACTIVE	0xF2	Receiver enable request could not be executed as in transmit state
MAC_ENUM_UNAVAILABLE_KEY	0xF3	Appropriate key is not available in ACL
MAC_ENUM_UNSUPPORTED_ATTRIBUTE	0xF4	PIB Set/Get on unsupported attribute

4.2 PAN Information Base

The PAN Information Base (PIB) consists of a number of parameters used by the MAC and Physical layers, which describe the Personal Area Network in which the node exists. The detailed use of these parameters is described in ref [1] section 7.4 and will not be dealt with further here. The mechanism that a network layer can use for reading (Get) and writing (Set) these parameters is described in the sections below

4.2.1 MAC Layer PIB access

This section describes how the MAC PIB parameters can be accessed.

4.2.1.1 MAC PIB parameters

The following table contains the PIB parameter name specified in ref [1] together with its data type and the range of values.

MAC PIB name	Type	Notes
eAckWaitDuration	Enum	Can take the following values MAC_PIB_ACK_WAIT_DURATION_HI (default) MAC_PIB_ACK_WAIT_DURATION_LO
bAssociationPermit	Boolean	Default value is FALSE
bAutoRequest	Boolean	Default value is TRUE
bBattLifeExt	Boolean	Default value is FALSE
eBattLifeExtPeriods	Enum	Can take the following values MAC_PIB_BATT_LIFE_EXT_PERIODS_HI (default) MAC_PIB_BATT_LIFE_EXT_PERIODS_LO
au8BeaconPayload	Uint8	Array of uint8 values of size u8BeaconPayloadLength
u8BeaconPayloadLength	Uint8	Maximum value is MAC_MAX_BEACON_PAYLOAD_LEN
u8BeaconOrder	Uint8	Range is MAC_PIB_BEACON_ORDER_MIN (0) MAC_PIB_BEACON_ORDER_MAX (15) (default)
u32BeaconTxTime	Uint32	Default value is 0
u8Bsn	Uint8	Beacon Sequence Number
sCoordExtAddr	MAC_ExtAddr_s	64bit Extended Address for the PAN Coordinator
u16CoordShortAddr	Uint16	16bit Short Address for the PAN Coordinator
u8Dsn	Uint8	Data Frame Sequence Number
bGtsPermit	Boolean	Default value is TRUE
u8MaxCsmaBackoffs_ReadOnly	Uint8	Range is MAC_PIB_MAX_CSMA_BACKOFFS_MIN (0) MAC_PIB_MAX_CSMA_BACKOFFS_MAX (5) Default is 4 and value cannot be set directly
u8MinBe_ReadOnly	Uint8	Range is MAC_PIB_MIN_BE_MIN (0) MAC_PIB_MIN_BE_MAX (3) Default is 3 and value cannot be set directly
u16PanId_ReadOnly	Uint16	16bit PAN ID
bPromiscuousMode_ReadOnly	Boolean	Default value is FALSE Value cannot be set directly
bRxOnWhenIdle_ReadOnly	Boolean	Default value is FALSE Value cannot be set directly

u16ShortAddr_ReadOnly	Uint16	16bit Short Address of device Cannot be set directly
u8SuperframeOrder	Uint8	Range is MAC_PIB_SUPERFRAME_ORDER_MIN (0) MAC_PIB_SUPERFRAME_ORDER_MAX (15) (default)
u16TransactionPersistenceTime	Uint16	Default value is 0x01F4
asAclEntryDescriptorSet	MAC_PibAclEntry_s	Array of structs defined in mac_pib.h
u8AclEntrySetSize	Uint8	Range is MAC_PIB_ACL_ENTRY_DESCRIPTOR_SET_SIZE_MIN (0) (default) MAC_PIB_ACL_ENTRY_DESCRIPTOR_SET_SIZE_MAX (15)
bDefaultSecurity	Boolean	Default value is FALSE
u8AclDefaultSecurityMaterialLength	Uint8	Range is MAC_PIB_ACL_DEFAULT_SECURITY_LEN_MIN (0) MAC_PIB_ACL_DEFAULT_SECURITY_LEN_MAX (26) Default value is 21
sDefaultSecurityMaterial	MAC_PibSecurityMaterial_s	Struct defined in mac_pib.h
u8DefaultSecuritySuite	Uint8	Range is MAC_PIB_DEFAULT_SECURITY_SUITE_MIN (0) (default) MAC_PIB_DEFAULT_SECURITY_SUITE_MAX (7)
u8SecurityMode	Uint8	Range is MAC_SECURITY_MODE_UNSECURED (default) MAC_SECURITY_MODE_ACL MAC_SECURITY_MODE_SECURED

4.2.1.2 PIB Handle

In order to access the PIB Attributes a handle to the PIB is required the Application gets a handle to the PIB with the following code:

```
/* At start of file */
#include "AppApi.h"
#include "mac_pib.h"

PRIVATE void *pvMac;
PRIVATE MAC_Pib_s *psPib;

/* Within application initialization function */
pvMac = pvAppApiGetMacHandle();
psPib = MAC_psPibGetHandle(pvMac);
```

Once the handle is obtained, PIB attributes can be read directly, e.g.

```
bMyAssociationPermit = psPib->bAssociationPermit;
```

Most of the PIB attributes can also be written using the PIB handle:

```
psPib->bAssociationPermit = bMyAssociationPermit;
```

However, the setting of some attributes needs to be done using auxiliary functions, as they also cause a change to hardware registers. The affected attributes and their associated functions are:

Attribute	Function to use when setting attribute
macMaxCSMABackoffs	MAC_vPibSetMaxCsmaBackoffs(void *pvMac, uint8 u8MaxCsmaBackoffs)
macMinBE	MAC_vPibSetMinBe(void *pvMac, uint8 u8MinBe)
macPANId	MAC_vPibSetPanId(void *pvMac, uint16 u16PanId)
macPromiscuousMode	MAC_vPibSetPromiscuousMode(void *pvMac, bool_t bNewState, FALSE)
macRxOnWhenIdle	MAC_vPibSetRxOnWhenIdle(void *pvMac, bool_t bNewState, FALSE)
macShortAddress	MAC_vPibSetShortAddr(void *pvMac, uint16 u16ShortAddr)

e.g.:

```
MAC_vPibSetShortAddr(pvMac, 0x1234);
```

4.2.1.3 MAC PIB Examples

The following is an example of writing the beacon order attribute in the PIB.

```
psPib->u8BeaconOrder = 5;
```

The following is an example of reading the coordinator short address from the PIB.

```
uint16 u16CoordShortAddr;
u16CoordShortAddr = psPib->u16CoordShortAddr;
```

The following is an example of writing to one of the variables within an access control list entry.

```
psPib->asAclEntryDescriptorSet[1].u8AclSecuritySuite = 0x01; /*AES-CTR*/
```

4.2.2 Physical Layer PIB access

This section describes how the PHY PIB parameters can be accessed.

4.2.2.1 Referencing PHY PIB parameters

The Physical Layer PIB Get and Set operations use codes to refer to the attribute that they are operating on. The following table contains the PIB attribute name specified in ref [1] together with its code number and the enumeration name defined by the MAC software, making up the type **`PHY_PibAttr_e`**.

PHY PIB name	Value	Enumeration name
phyCurrentChannel	0x00	PHY_PIB_ATTR_CURRENT_CHANNEL
phyChannelsSupported	0x01	PHY_PIB_ATTR_CHANNELS_SUPPORTED
phyTransmitPower	0x02	PHY_PIB_ATTR_TX_POWER
phyCCAMode	0x03	PHY_PIB_ATTR_CCA_MODE

Pre-defined values are available for these PHY PIB attributes, as specified in the following table:

Attribute	Values
PHY_PIB_ATTR_CURRENT_CHANNEL	PHY_PIB_CURRENT_CHANNEL_DEF (default - 11) PHY_PIB_CURRENT_CHANNEL_MIN (minimum - 11) PHY_PIB_CURRENT_CHANNEL_MAX (maximum - 26)
PHY_PIB_ATTR_CHANNELS_SUPPORTED	PHY_PIB_CHANNELS_SUPPORTED_DEF (default - 0x07fff800)
PHY_PIB_ATTR_TX_POWER	PHY_PIB_TX_POWER_DEF (default - 0x40) PHY_PIB_TX_POWER_MIN (minimum - 0) PHY_PIB_TX_POWER_MAX (maximum - 0xbf) PHY_PIB_TX_POWER_MASK (0x3f) {mask to be used with dB settings below} PHY_PIB_TX_POWER_1DB_TOLERANCE (0x00) PHY_PIB_TX_POWER_3DB_TOLERANCE (0x40) PHY_PIB_TX_POWER_6DB_TOLERANCE (0x80)
PHY_PIB_ATTR_CCA_MODE	PHY_PIB_CCA_MODE_DEF (default - 1) PHY_PIB_CCA_MODE_MIN (minimum - 1) PHY_PIB_CCA_MODE_MAX (maximum - 3)

Both the Get and Set operations return a `PHY_Enum_e` enumeration status value to indicate success or failure of the operation. This enumeration is defined in Table 16 (section 6.3.6) of the 802.15.4 specification (d18). Refer to the specification for the definitions.

Name	Value	Description
PHY_ENUM_INVALID_PARAMETER	0x05	A SET/GET request was issued with a parameter in the primitive that is outside the valid range.
PHY_ENUM_SUCCESS	0x07	A SET/GET operation was successful.
PHY_ENUM_UNSUPPORTED_ATTRIBUTE	0x0A	A SET/GET request was issued with the identifier of an attribute that is not supported.

4.2.2.2 eAppApiPlmeGet

PHY PIB parameter values can be returned to the network layer using the *eAppApiPlmeGet* routine. The Get request routine is defined as follows:

Declaration

```
PUBLIC PHY_Enum_e eAppApiPlmeGet (PHY_PibAttr_e ePhyPibAttribute,
                                  uint32 *pu32PhyPibValue)
```

Inputs

`PHY_PibAttr ePhyPibAttribute`	Enumeration defining which attribute to access
`uint32 *pu32PhyPibValue`	Pointer to a location used to hold the result of the Get operation.

Outputs

`PHY_Enum_e`	Enumeration status value that indicates success or failure of the operation.

Description

This routine can be used to retrieve the current value of one of the PHY PIB attributes.

If the routine returns PHY_ENUM_SUCCESS, the value of the PIB PHY attribute retrieved has been copied into the location pointed to by `pu32PhyPibValue`.

4.2.2.2.1 Example

The following example illustrates how to read the current channel:

```
uint32 u32sChannel;
if (eAppApiPlmeGet (PHY_PIB_ATTR_CURRENT_CHANNEL,&u32sChannel)
     == PHY_ENUM_SUCCESS)
{
    printf("Channel is %d\n", u32Channel);
}
```

4.2.2.3 eAppApiPlmeSet

PHY PIB parameter values can be changed by the network layer using the *PLME-Set.request* primitive. The request is sent using the **`eAppApiPlmeSet`** routine. The Set request routine is defined as follows:

Declaration

```
PUBLIC PHY_Enum_e eAppApiPlmeSet (PHY_PibAttr ePhyPibAttribute,
                                  uint32 u32PhyPibValue)
```

Inputs

`PHY_PibAttr ePhyPibAttribute`	Enumeration defining which parameter to access
`uint32 pu32PhyPibValue`	The uint32 value the parameter will be set to.

Outputs

`PHY_Enum_e`	Enumeration status value that indicates success or failure of the operation.

Description

This routine can be used to change the value of one of the PHY PIB parameters.

If the routine returns PHY_ENUM_SUCCESS, the value of the PIB PHY parameter has been changed to `u32PhyPibValue`.

4.2.2.3.1 Examples

The following example illustrates how to set the current channel:

```
if (eAppApiPlmeSet(PHY_PIB_ATTR_CURRENT_CHANNEL, u8Channel) !=
        PHY_ENUM_SUCCESS)
{
        // Handle error;
}
```

The following example illustrates how to set the transmit power to 0 dBm:

```
if (eAppApiPlmeSet(PHY_PIB_ATTR_TX_POWER, 0) != PHY_ENUM_SUCCESS)
{
        // Handle error;
}
```

4.3 MAC Reset

The MAC and PHY can be reset by the network layer (i.e. return all variables to a default value and disable the transmitter of the PHY) to get it into a known state before issuing further MAC requests. The PIB may be reset to its default values by the request, or it may retain its current data.

A reset request is sent using the **vAppApiMlmeRequest()** routine. The request structure is defined as follows:

```
/*
 * MAC reset request. Use type MAC_MLME_REQ_RESET
 */
typedef struct
{
    uint8 u8SetDefaultPib;
} MAC_MlmeReqReset_s;
```

The field **u8SetDefaultPib** controls whether the PIB contents are to be reset to their default values. When set to TRUE the PIB is reset.

The confirm is generated synchronously and contains the following structure giving the result of the reset request

```
/* Structure for MLME-RESET.confirm
 *
 */
typedef struct
{
    uint8 u8Status;
} MAC_MlmeCfmReset_s;
```

The status field of the confirm may take the values **MAC_ENUM_SUCCESS** indicating that the reset took place, or **MAC_ENUM_DISABLE_TRX_FAILURE** which shows that the transmitter or receiver of the node could not be turned off.

4.3.1 Reset Example

The following is an example of using the reset request.

```
/* Structures used to hold data for MLME request and response */
MAC_MlmeReqRsp_s   sMlmeReqRsp;
MAC_MlmeSyncCfm_s  sMlmeSyncCfm;

/* Request Reset */
sMlmeReqRsp.u8Type = MAC_MLME_REQ_RESET;
sMlmeReqRsp.u8ParamLength = sizeof(MAC_MlmeReqReset_s);
sMlmeReqRsp.uParam.sReqReset.u8SetDefaultPib = TRUE; /* Reset PIB */

vAppApiMlmeRequest(&sMlmeReqRsp, &sMlmeSyncCfm);

/* Handle synchronous confirm */
if (sMlmeSyncCfm.u8Status != MAC_MLME_CFM_OK)
{
    /* Error during MLME-Reset */
}
```

4.4 Scan

The MAC supports the scan feature as defined in ref [1] section 7.1.11 and 7.5.2

Any scan request will cause other activities, which use the transceiver to shut down for the duration of the scan period. This means that beacon transmission is suspended when a coordinator begins scanning, and will resume at the end of the scan period. The Application or NWK layer above the MAC is responsible for initiating a scan at the appropriate time in order not to cause problems with other activities. The NWK/Application is also responsible for ensuring that scans are requested over channels supported by the PHY, and that only those scan types the device supports are requested.

All scans require the Application/NWK to supply a set of channels to be scanned, and a duration over which the measurement on a channel will take place. The total scan time will be the time spent measuring each requested channel for its scan duration, up to a limit of **MAC_MAX_SCAN_CHANNELS** (16) channels

4.4.1 Energy Detect Scan

Energy Detect scan is not supported on RFDs. When this scan is requested, the MAC will measure the energy on each of the channels requested or until it has measured MAC_MAX_SCAN_CHANNELS channels. Used during PAN initialisation where the coordinator is trying to find the clearest channel on which to begin setting up a PAN.

4.4.2 Active Scan

The MAC tunes to each requested channel in turn and sends a beacon request to which all coordinators on that channel should respond by sending a beacon, even if not generating beacons in normal operation. For each unique beacon received, the MAC stores the PAN details in a PAN descriptor which is returned in the *MLME-Scan.confirm* primitive for the scan request. A total of **MAC_MAX_SCAN_PAN_DESCRS** (8) entries may be carried in the Scan Confirm primitive. Scanning terminates either when all channels specified have each been scanned for the duration requested, or after **MAC_MAX_SCAN_PAN_DESCRS** unique beacons have been found, whether or not all requested channels have been scanned.

4.4.3 Passive Scan

For a Passive scan the MAC tunes to each requested channel in turn and listens for a beacon transmission for a period specified in the *MLME-Scan.request.* For each unique beacon received, the MAC stores the PAN details in a PAN descriptor which is returned in the *MLME-Scan.confirm* primitive corresponding to the *MLME-Scan.request*. A total of **MAC_MAX_SCAN_PAN_DESCRS** (8) entries may be carried in the *MLME-Scan.confirm* message. Scanning terminates either when all channels specified have each been scanned for the duration requested, or after **MAC_MAX_SCAN_PAN_DESCRS** unique beacons have been found, whether or not all requested channels have been scanned.

4.4.4 Orphan Scan

An orphan scan can be performed by a device which has lost synchronisation with its coordinator. The device requests an orphan scan using the *MLME-Scan.request* primitive with the scan type set to orphan. For each channel specified the device tunes to the channel and then sends an orphan notification message. It then waits on the channel in receive mode until it receives a coordinator realignment command or when **MAC_RESPONSE_WAIT_TIME** superframe

periods have passed. If a coordinator realignment command is seen the scan will be terminated and the *MLME-Scan.confirm* status will be **MAC_ENUM_SUCCESS**. The contents of the realignment command are used to update the PIB (macCoordShortAddress, macPANId, macShortAddress). If all the requested channels are scanned and no coordinator realignment command is seen, the *MLME-Scan.confirm* status will be **MAC_ENUM_NO_BEACON.**

4.4.5 Scan Request

A scan is requested using the *MLME-Scan.request* primitive. The request is sent using the **vAppApiMlmeRequest()** routine. The request structure is defined as follows:

```
typedef struct
{
    uint32    u32ScanChannels;   /* Scan channels bitmap */
    uint8     u8ScanType;        /* type of scan to be requested */
    uint8     u8ScanDuration;    /* Scan duration */
} MAC_MlmeReqScan_s;
```

u8ScanType contains the type of scan to be requested, as specified in enumeration **MAC_MlmeScanType_e** shown below:

```
typedef enum
{
    MAC_MLME_SCAN_TYPE_ENERGY_DETECT = 0,   /* Energy detect scan */
    MAC_MLME_SCAN_TYPE_ACTIVE = 1,          /* Active scan */
    MAC_MLME_SCAN_TYPE_PASSIVE = 2,         /* Passive scan */
    MAC_MLME_SCAN_TYPE_ORPHAN = 3,          /* Orphan scan */
    NUM_MAC_MLME_SCAN_TYPE
} MAC_MlmeScanType_e;
```

u32ScanChannels is a bitmap of 32 channels which may be scanned.

Only channels 26-11 are available with the 2.45 GHz PHY. Channels 31-27 are reserved.

Bits representing the channels to be scanned are set to 1.

U8ScanDuration may take the values 0 – 14 and represents the time to scan a channel measured in superframe periods (1 superframe time = 960 symbols) where the number of superframes is specified as $(2^n + 1)$ where n is u8ScanDuration.

4.4.6 Scan Confirm

Results from a *MLME-Scan.request* primitive are conveyed back asynchronously in the *MLME-Scan.confirm* primitive using the callback routines registered at system start in the call **u32AppApiInit()**. They may also be sent synchronously to the Application/NWK layer as part of the **vAppApiMlmeRequest()** used to send the Scan Request. The structure is defined below:

```
typedef
{
    uint8  u8Status;                /* Status of scan request */
    uint8  u8ScanType;              /* Scan type */
    uint8  u8ResultListSize;        /* Size of scan results list */
    uint8  u8Pad;                   /* Alignment */
    uint32 u32UnscannedChannels];   /* Bitmap of unscanned channels */
    MAC_ScanList_u uList;           /* Scan results list */
} MAC_MlmeCfmScan_s;
```

u8Status returns the result of the associated scan request. This may take the value MAC_ENUM_SUCCESS if the scan found one or more PANs in the case of an Energy Detect, Passive or Active scan, or MAC_ENUM_NO_BEACON if no beacons were seen during an orphan scan.

u8ScanType contains the same value as the corresponding field in the *MLME-Scan.request* primitive to show the type of scan performed.

u32UnscannedChannels contains a bitmap of the channels specified in the request which were not scanned during the scanning process. The mapping of channel to bit is as for the corresponding request and unscanned channels are denoted by being set to 1.

u8ResultListSize is the size in bytes of the result list from the scan. If the **u8ScanType** value is **MAC_MLME_SCAN_TYPE_ORPHAN** the value of this field will be 0.

uList is a union containing either the results of an energy detect scan in the byte array **au8EnergyDetect** or the results of detecting beacons during an Active or Passive scan in the PAN descriptor array **asPanDescr**

```
typedef union
{
    uint8           au8EnergyDetect[MAC_MAX_SCAN_CHANNELS];
    MAC_PanDescr_s  asPanDescr[MAC_MAX_SCAN_PAN_DESCRS];
} MAC_ScanList_u;
```

A PAN descriptor structure contains the following information:

```
Typedef struct
{
    MAC_Addr_s sCoord;                  /* Coordinator address */
    uint8      u8LogicalChan;           /* Logical channel */
    uint8      u8GtsPermit;             /* True if beacon is from PAN
                                         * coordinator which accepts GTS
                                         * requests
                                         */
    uint8      u8LinkQuality;           /* Link quality of the received
                                         * beacon
                                         */
    uint8      u8SecurityUse;           /* True if beacon received was
                                         * secure
                                         */
    uint8      u8AclEntry;              /* Security mode used in ACL
                                         * entry
                                         */
    uint8      u8SecurityFailure;       /* True if there was an error in
                                         * security processing
                                         */
    uint16     u16SuperframeSpec;       /* Superframe specification */
    uint32     u32TimeStamp;            /* Timestamp of the received
                                         * beacon
                                         */
} MAC_PanDescr_s;
```

sCoord is a structure, which holds the MAC address of the coordinator, which transmitted the beacon. It consists of the following fields:

```
typedef struct
{
    uint8      u8AddrMode;                  /* Address mode */
    uint16     u16PanId;                    /* PAN ID */
    MAC_Addr_u uAddr;                       /* Address */
} MAC_Addr_s;
```

u8AddrMode denotes the type of addressing used to specify the address of the coordinator and may take the following values:

Addressing mode value	Description
0	PAN identifier and address field are not present
1	Reserved
2	Address field contains 16-bit short address
3	Address field contains 64-bit extended address

If **u8AddrMode** is non-zero then the following fields contain the PAN identifier and either the short or the extended address of the coordinator sending the beacon

u16PanId is a the PAN identifier

uAddr is a union, which may contain either the 16-bit short address or the 64-bit extended address

```
typedef union
{
    uint16           u16Short;              /* Short address */
    MAC_ExtAddr_s    sExt;                  /* Extended address */
} MAC_Addr_u;
```

The 64 bit extended address is held in a MAC_ExtAddr_s as follows;

```
typedef struct
{
    uint32 u32L;  /* Low word */
    uint32 u32H;  /* High word */
} MAC_ExtAddr_s;
```

u8LogicalChan holds the channel number on which the beacon was transmitted. For the 2.45GHz PHY this field may take the values 11 to 26 corresponding to the allowed channel numbers for the radio.

u8GtsPermit is set to 1 if the beacon is from a PAN coordinator, which accepts GTS (Guaranteed Time Slot) requests.

u8LinkQuality carries a measure of the quality of the transmission which carried the beacon ranging from 0 to 255, 0 representing low quality.

u8SecurityUse is set to 1 if the beacon is using security, 0 otherwise.

u8AclEntry contains the value of the security mode in use by the sender of the beacon, as retrieved from the ACL entry corresponding to the beacon sender and may take the values 0 to 7 denoting the security suite in use. If the sender is not found in the ACL this value is set to 8.

The security modes are defined as

Value	Mode
0	None
1	AES-CTR
2	AES-CCM-128
3	AES-CCM-64
4	AES-CCM-32
5	AES-CBC-MAC-128
6	AES-CBC-MAC-64
7	AES-CBC-MAC-32

u8SecurityFailure is set to 1 if there was an error during the security processing of the beacon, 0 otherwise. Always 0 if **u8SecurityUse** is 0.

u16SuperframeSpec contains information about the superframe used in the PAN that this beacon describes. It follows the same format as that specified in section 7.2.2.1.2 of ref [1].

u32TimeStamp indicates the time that the beacon was received, measured in symbol periods.

4.4.7 Orphan Indication

An Orphan Indication is generated by the MAC of a coordinator to its Application/NWK layer to indicate that it has received an orphan notification message transmitted by an orphan node. The indication message is sent to the Application/NWK layer using the callback routines registered at system start in the call **u32AppApiInit()**. The structure of the Orphan Indication is as follows:

```
typedef struct
{
    MAC_ExtAddr_s sDeviceAddr;  /* Extended address of orphaned device*/
    uint8         u8SecurityUse;/* True if security was used on command
                                 * frames
                                 */
    uint8         u8AclEntry;   /* Security suite used */
} MAC_MlmeIndOrphan_s;
```

sDeviceAddr contains the full 64-bit extended address of the orphaned node.

u8SecurityUse indicates if security was being used when the orphan notification was sent (set to 1 if this is true).

u8AclEntry is the security mode (values o to 7) being used by the node transmitting the orphan notification as stored in the coordinators ACL for that address. If the orphan node cannot be found in the ACL the value is set to 8.

4.4.8 Orphan Response

An Orphan Response is generated by the Application/NWK layer in response to receiving an Orphan Indication. The response is sent using the **vAppApiMlmeRequest()** routine. It contains the following fields

```
typedef struct
{
    MAC_ExtAddr_s sOrphanAddr;/* Orphaned Device's extended address */
    uint8      u16OrphanShortAddr;
                              /* Short addr Orphaned Device should use /
    uint8      u8Associated; /* True if Device was previously associated
                              */
    uint8      u8SecurityEnable;
                             /* True if security is to be used on
                              * command frames
                              */
} MAC_MlmeRspOrphan_s;
```

sOrphanAddr carries the full 64-bit extended address of the orphan node, as carried in the Orphan Indication.

u16OrphanShortAddr holds the 16-bit short address that the orphan node used within the PAN if it was previously associated, and should continue to use. If the node was not previously associated with the coordinator, the value 0xFFFF is returned. If the node is not to use a short address, then the value 0xFFFE is returned in this field.

u8Associated if set to 1 indicates that the node was previously associated with this coordinator.

u8SecurityEnable should be set to 1 if the orphan node is to use security processing on its communication with the coordinator, or 0 otherwise.

On receiving this response, if the orphan was previously associated with the coordinator, the MAC will send a coordinator realignment command to the orphan. The result of sending this command will be to generate a *MLME-COMM-STATUS.indication* from the MAC to the Application/NWK layer. See section 4.4.9 Comm Status Indication for the usage of this primitive.

4.4.9 Comm Status Indication

A Comm Status Indication is generated by the MAC to inform the Application/NWK layer of a coordinator the result of a communication with another node triggered by a previous primitive (*MLME-Orphan.response* and *MLME-Associate.response*). The indication message is sent to the Application/NWK layer using the callback routines registered at system start in the call **u32AppApiInit()**. The structure of the Comm Status Indication is as follows:

```
typedef struct
{
    MAC_Addr_s sSrcAddr;  /* Source address of frame */
    MAC_Addr_s sDstAddr;  /* Destination address of frame */
    uint8      u8Status;  /* Status of communication */
} MAC_MlmeIndCommStatus_s;
```

sSrcAddr and **sDstAddr** contain the addresses of the source and destination of the frame, their formats being shown below

```
typedef struct
{
    uint8      u8AddrMode;  /* Address mode */
    uint16     u16PanId;    /* PAN ID */
    MAC_Addr_u uAddr;       /* Address */
} MAC_AddrOnly_s;
```

u8AddrMode denotes the type of addressing used to specify the address and may take the following values:

Addressing mode value	Description
0	No address – address field **uAddr** omitted
1	Reserved
2	Address field contains 16-bit short address
3	Address field contains 64-bit extended address

uAddr is a union which can contain either a 64-bit extended address or a 16-bit short address.

u16PanId is the PAN id of the network.

u8Status is the result of the transaction whose status is being reported, and takes on values from the enumeration **MAC_enum_e.** In the case of an Orphan Response the possible results are

Status	Reason
MAC_ENUM_UNAVAILABLE_KEY	couldn't find a security key in the ACL for the transmission
MAC_ENUM_FAILED_SECURITY_CHECK	failure during security processing of the frame
MAC_ENUM_CHANNEL_ACCESS_FAILURE	couldn't get access to the radio channel to perform the transmission
MAC_ENUM_NO_ACK	didn't get an acknowledgement from the orphan node after sending the coordinator realignment command
MAC_ENUM_INVALID_PARAMETER	invalid parameter value or parameter not supported in the Orphan Response
MAC_ENUM_SUCCESS	coordinator realignment command sent successfully

4.4.10 Examples

The following is an example of performing an active channel scan (see the next example for details of handling the deferred confirm that is generated by this request).

```
#define CHANNEL_BITMAP              0x7800

/* Structures used to hold data for MLME request and response */
MAC_MlmeReqRsp_s   sMlmeReqRsp;
MAC_MlmeSyncCfm_s  sMlmeSyncCfm;

/* Request active channel scan */
sMlmeReqRsp.u8Type = MAC_MLME_REQ_SCAN;
sMlmeReqRsp.u8ParamLength = sizeof(MAC_MlmeReqScan_s);
sMlmeReqRsp.uParam.sReqScan.u8ScanType = MAC_MLME_SCAN_TYPE_ACTIVE;
sMlmeReqRsp.uParam.sReqScan.u32ScanChannels = CHANNEL_BITMAP;
sMlmeReqRsp.uParam.sReqScan.u8ScanDuration = 6;

vAppApiMlmeRequest(&sMlmeReqRsp, &sMlmeSyncCfm);


/* Handle synchronous confirm */
if (sMlmeSyncCfm.u8Status != MAC_MLME_CFM_DEFERRED)
{
    /* Unexpected result: scan request should result in a deferred
       confirmation (i.e. we will receive it later) */
}
```

The following is an example of handling a deferred active scan confirmation (assumes data is passed as a pointer to a deferred confirm indicator data type i.e. `MAC_MlmeDcfmInd_s *psMlmeInd`.)

```
#define DEMO_PAN_ID                         0x0e1c
#define DEMO_COORD_ADDR                     0x0e00

MAC_PanDescr_s *psPanDesc;
int i;

if (psMlmeInd->u8Type == MAC_MLME_DCFM_SCAN)
{
    if ((psMlmeInd->uParam.sDcfmScan.u8Status == MAC_ENUM_SUCCESS)
       && (psMlmeInd->uParam.sDcfmScan.u8ScanType ==
           MAC_MLME_SCAN_TYPE_ACTIVE))
    {
        /* Determine which, if any, network contains demo coordinator.
           Algorithm for determining which network to connect to is
           beyond the scope of 802.15.4, and we use a simple approach
           of matching the required PAN ID and short address, both of
           which we already know */

        i = 0;
        while (i < psMlmeInd->uParam.sDcfmScan.u8ResultListSize)
        {
           psPanDesc = &psMlmeInd->uParam.sDcfmScan.uList.asPanDescr[i];
```

```
                if ((psPanDesc->sCoord.u16PanId == DEMO_PAN_ID)
                   && (psPanDesc->sCoord.u8AddrMode == 2)
                   && (psPanDesc->sCoord.uAddr.u16Short == DEMO_COORD_ADDR))
                {
                    /* Matched so start to synchronise and associate */
                }
            }
        }
    }
```

The following is an example of requesting an energy detection scan.

```
/* Structures used to hold data for MLME request and response */
MAC_MlmeReqRsp_s   sMlmeReqRsp;
MAC_MlmeSyncCfm_s  sMlmeSyncCfm;

/* Request active channel scan */
sMlmeReqRsp.u8Type = MAC_MLME_REQ_SCAN;
sMlmeReqRsp.u8ParamLength = sizeof(MAC_MlmeReqScan_s);

sMlmeReqRsp.uParam.sReqScan.u8ScanType =
                                        MAC_MLME_SCAN_TYPE_ENERGY_DETECT;

sMlmeReqRsp.uParam.sReqScan.u32ScanChannels = ALL_CHANNELS_BITMAP;
sMlmeReqRsp.uParam.sReqScan.u8ScanDuration = 6;

vAppApiMlmeRequest(&sMlmeReqRsp, &sMlmeSyncCfm);

/* Check immediate response */
if (sMlmeSyncCfm.u8Status != MAC_MLME_CFM_DEFERRED)
{
    /* Unexpected result: scan request should result in a deferred
       confirmation (i.e. we will receive it later) */
}
```

The following is an example of handling the response (a deferred confirmation) to an energy detect scan. Assumes data is passed as a pointer to a deferred confirm indicator data type i.e. `MAC_MlmeDcfmInd_s *psMlmeInd`.

```
int i;
uint8 u8ClearestChan, u8MinEnergy;
uint8 *pu8EnergyDetectList;

if (psMlmeInd->u8Type == MAC_MLME_DCFM_SCAN)
{
    /* Check that this response is the result of a
       successful energy detect scan */

    if ((psMlmeInd->uParam.sDcfmScan.u8Status == MAC_ENUM_SUCCESS)
       && (psMlmeInd->uParam.sDcfmScan.u8ScanType ==
          MAC_MLME_SCAN_TYPE_ENERGY_DETECT))
    {
        u8MinEnergy = 0xff;
        u8ClearestChan = 11;
        pu8EnergyDetectList =
```

```
            psMlmeInd->uParam.sDcfmScan.uList.au8EnergyDetect;

            /* Find clearest channel (lowest energy level). Assumes
               that all 16 channels available to 2.4GHz band have
               been scanned. */

            for (i = 0; i < MAC_MAX_SCAN_CHANNELS; i++)
            {
                if (pu8EnergyDetectList[i] < u8MinEnergy)
                {
                    u8MinEnergy = pu8EnergyDetectList[i];
                    u8ClearestChan = i + 11;
                }
            }
        }
    }
```

4.5 Start

The MAC supports the PAN start feature as defined in ref [1] section 7.1.14 and 7.5.2.

The Start feature is used by a FFD to begin acting as the coordinator of a new PAN or to begin transmitting beacons when associated with a PAN. A PAN should only be started after an Active Scan has been performed in order to find which PAN identifiers are currently in use. A PAN is started using the *MLME-START.request* primitive.

4.5.1 Start request

Beacon generation is requested using the *MLME-Start.request* primitive. The request is sent using the **`vAppApiMlmeRequest()`** routine. The request structure is defined as follows:

```
typedef struct
{
    uint16 u16PanId;
                              /* The PAN ID indicated in the beacon */
    uint8  u8Channel;         /* Channel to send beacon out on */
    uint8  u8BeaconOrder;     /* Beacon order */
    uint8  u8SuperframeOrder; /* Superframe order */
    uint8  u8PanCoordinator;  /* True for a PAN Coordinator */
    uint8  u8BatteryLifeExt;  /* True if battery life extension timings
                               * are to be used
                               */
    uint8  u8Realignment;     /* True if Coordinator realignment is sent
                               * when superframe parameters change
                               */
    uint8  u8SecurityEnable;   /* True if security is to be used on
                               * command frames
                               */
} MAC_MlmeReqStart_s;
```

`u16PanId` contains the 16-bit PAN identifier as selected by the Application/NWK layer.

`u8Channel` carries the logical channel number (11 to 26 for 2.45 GHz PHY) on which the beacon will be transmitted.

u8BeaconOrder defines how often a beacon will be transmitted. It takes values 0-15, 0-14 being used to define the beacon interval, which is calculated as 2**BO times the base superframe duration (number of symbols in superframe slot x number of slots in superframe = 960 symbols). If the value is 15, beacons are not transmitted and the Superframe order parameter is ignored.

u8SuperframeOrder defines how long the active period of the superframe is including the beacon period. Its value can be from 0 to BeaconOrder as specified above or 15. The active period time is specified as 2**SO times the base superframe duration. If the value is 15, the superframe will not be active after the beacon.

u8PanCoordinator is set to TRUE if the FFD is to become the PAN coordinator for a new PAN, otherwise if set to FALSE the FFD will transmit beacons on the existing PAN with which it is associated.

u8BatteryLifeExt if set to TRUE allows for battery life extension to be used by turning off the receiver of the FFD for a part of the contention period after the beacon is transmitted. If set to FALSE the receiver remains enabled for the whole of the contention access period after the beacon.

u8Realignment if set to TRUE will cause a coordinator realignment command to be broadcast prior to changing the superframe settings in order to alert the nodes in the PAN of the change. Set to FALSE otherwise.

u8SecurityEnable is set to TRUE if security is used on beacon frames, or false otherwise.

4.5.2 Start confirm

A *MLME-Start.confirm* primitive is generated by the MAC to inform the Application/NWK layer of the results of a *MLME-Start.request*. The confirm message is sent to the Application/NWK layer using the callback routines registered at system start in the call **u32AppApiInit()**. It may also be sent synchronously to the Application/NWK layer as part of the **vAppApiMlmeRequest()** used to send the Start Request. The structure of the Start Confirm is as follows:

```
typedef struct
{
    uint8 u8Status; /* Status of superframe start request */
} MAC_MlmeCfmStart_s;
```

u8Status contains the result of the corresponding *MLME-Start.request* primitive. It takes values from the **MAC_enum_e** enumeration type as follows:

Value	Reason
MAC_ENUM_NO_SHORT_ADDRESS	The PIB value for the short address is set to 0xFFFF
MAC_ENUM_UNAVAILABLE_KEY	The u8SecurityEnable field of the request is set o TRUE but the key and security information for the broadcast address cannot be obtained from the ACL in the PIB
MAC_ENUM_FRAME_TOO_LONG	The security encoding process on a beacon results in a beacon which is longer than the maximum MAC frame size
MAC_ENUM_FAILED_SECURITY_CHECK	For any other reason than the above that security processing fails
MAC_ENUM_INVALID_PARAMETER	For any parameter out of range or not supported
MAC_ENUM_SUCCESS	Start primitive was successful

4.5.3 Examples

The following is an example of a typical start request.

```
#define DEMO_PAN_ID        0x1234

/* Structures used to hold data for MLME request and response */
MAC_MlmeReqRsp_s sMlmeReqRsp;
MAC_MlmeSyncCfm_s sMlmeSyncCfm;

/* Start beacons */
sMlmeReqRsp.u8Type = MAC_MLME_REQ_START;
sMlmeReqRsp.u8ParamLength = sizeof(MAC_MlmeReqStart_s);
sMlmeReqRsp.uParam.sReqStart.u16PanId = DEMO_PAN_ID;
sMlmeReqRsp.uParam.sReqStart.u8Channel = 11;

/* Eight beacons per second */
sMlmeReqRsp.uParam.sReqStart.u8BeaconOrder = 3;

/* Only receive during first half of superframe: save energy */
sMlmeReqRsp.uParam.sReqStart.u8SuperframeOrder = 2;
sMlmeReqRsp.uParam.sReqStart.u8PanCoordinator = TRUE;
sMlmeReqRsp.uParam.sReqStart.u8BatteryLifeExt = FALSE;
sMlmeReqRsp.uParam.sReqStart.u8Realignment = FALSE;
sMlmeReqRsp.uParam.sReqStart.u8SecurityEnable = FALSE;

vAppApiMlmeRequest(&sMlmeReqRsp, &sMlmeSyncCfm);

/* Handle synchronous confirm */
if (sMlmeSyncCfm.u8Status != MAC_MLME_CFM_OK)
{
    /* Error during MLME-Start */
}
```

4.6 Synchronisation

The MAC supports the Synchronisation feature as defined in ref [1] section 7.1.14 and 7.5.4

The purpose of the synchronisation feature is to allow devices to synchronise to beacon transmissions from PAN coordinators in order to be able to receive pending data held at the coordinator. Where a PAN does not perform beacon transmission, data synchronisation is performed by the device polling the PAN coordinator. A device can only acquire synchronisation to a beacon in the PAN in which it is associated; on receiving a beacon it can either track the beacon, turning on its receiver at some point before the beacon is due to be transmitted or it may receive a single beacon and then not attempt to receive any others.

Synchronisation is initiated using the *MLME_SYNC.request* primitive, which starts a search for a beacon. During the beacon search the device listens for a beacon for a time ((2**n) +1) base superframes (base superframe duration is 960 symbols) where n is the beacon order contained in the PIB. The search is repeated **`MAC_MAX_LOST_BEACONS`** (4) times and if a beacon is not found at the end of this search the Sync loss indication is issued.

If a previously synchronised device, which is tracking a beacon, misses **`MAC_MAX_LOST_BEACONS`** (4) consecutive beacons, synchronisation has been lost and a Sync Loss indication is issued.

Synchronisation is also lost if a PAN identifier conflict is detected, either by a coordinator receiving a beacon with the PAN coordinator indicator set and the same PANid that it is using, or receiving a PAN ID conflict notification from a device, or a device receiving a beacon with the PAN coordinator indicator set, the same PANid it expects but from a different coordinator.

In the latter case, the device transmits a PAN ID conflict notification message to its PAN coordinator. The Sync Loss indication will be issued.

If a beacon is received that uses security, and an error occurs when it is being processed, the MAC generates a *MLME-COMM-STATUS.indication* to the Application/NWK layer (see 4.4.9 Comm Status Indication) with a status of **`MAC_ENUM_FAILED_SECURITY_CHECK`**.

If a valid beacon is received (i.e. comes from the correct PAN coordinator address and has the correct PANid) a Beacon Notify indication is generated by the MAC to Application/NWK layer. Depending on the setting of **`MAC_PIB_ATTR_AUTO_REQUEST`** in the PIB the MAC may start to extract pending data from the coordinator.

For non-beaconing PANs, devices can extract pending data from the coordinator by issuing a *MLME-POLL.request* and the presence of data will be returned in the corresponding *MLME-POLL.confirm*, together with the actual data in a *MCPS-DATA.indication* primitive

4.6.1 Sync request

The *MLME-SYNC.request* primitive is used to tell the MAC to attempt to acquire a beacon. The request is sent to the MAC using the **`vAppApiMlmeRequest()`** routine. The request structure is defined as follows:

```
typedef struct
{
    uint8  u8Channel;       /* Channel to listen for beacon on */
    uint8  u8TrackBeacon;   /* True if beacon is to be tracked */
} MAC_MlmeReqSync_s;
```

u8Channel contains the logical channel on which the MAC will use to try to find beacon transmissions. For the 2.45 GHz PHY this field will take on values of 11 to 26

u8TrackBeacon is set to TRUE if the device is to continue tracking beacon transmissions following reception of the first beacon. Set to FALSE otherwise.

4.6.2 Sync loss indication

The sync loss indication is used to show to the Application/NWK layer that there has been a loss of synchronisation with the beacon, either because a beacon could not be found when a beacon search is initiated by a *MLME-SYNC.request*, or because a previously synchronised device tracking the beacon. The indication message is sent to the Application/NWK layer using the callback routines registered at system start in the call **u32AppApiInit()**. The structure of the Sync Loss is as follows:

```
typedef struct
{
    uint8 u8Reason; /* Synchronisation loss reason */
} MAC_MlmeIndSyncLoss_s;
```

u8Reason is the reason for the loss of synchronisation and takes a value from the **MAC_enum_e** enumeration

Value	Reason
MAC_ENUM_PAN_ID_CONFLICT	Generated when a device detects a PAN id conflict
MAC_ENUM_REALIGNMENT	A coordinator realignment command was received and the device is not performing an Orphan Scan
MAC_ENUM_BEACON_LOST	Failed to see MAC_MAX_LOST_BEACONS consecutive beacons either when tracking transmissions or searching for beacons after a Sync request

4.6.3 Beacon Notify Indication

A Beacon Notify Indication is generated by the MAC to inform the Application/NWK layer that a beacon transmission has been received. The indication message is sent to the Application/NWK layer using the callback routines registered at system start in the call **u32AppApiInit()**. The structure of the Beacon Notify Indication is as follows:

```
typedef struct
{
    MAC_PanDescr_s   sPANdescriptor;  /* PAN descriptor */
    uint8            u8BSN;           /* Beacon sequence number */
    uint8            u8PendAddrSpec;  /* Pending address specification
                                          */
    uint8            u8SDUlength;     /* Length of following payload */
    MAC_Addr_u       uAddrList[7];    /* Pending addresses */
    uint8            u8SDU[MAC_MAX_BEACON_PAYLOAD_LEN];
                                      /* Beacon payload */
} MAC_MlmeIndBeacon_s;
```

sPANdescriptor holds the information about the PAN that the beacon carries. This structure has already been described in PAN descriptor.

u8BSN contains the Beacon Sequence Number, which can take the value 0 to 255.

u8PendAddrSpec consists of a byte, which encodes the number of nodes, which have messages pending at the coordinator, which generated the beacon. There are at most seven nodes which can be shown as having messages stored at the coordinator although there may be more messages actually stored. The Address Specification may contain a mixture of short and extended addresses, up to the total of 7. It is encoded as follows:

Bits 0..2	3	4..6	7
Number of short addresses pending	Reserved	Number of extended addresses pending	Reserved

u8SDUlength contains the length in bytes of the beacon payload field, up to a maximum of **MAC_MAX_BEACON_PAYLOAD_LEN**

uAddrList contains an array of seven short or extended addresses corresponding to the numbers in **u8PendAddrSpec**. The addresses are ordered so that all the short addresses are listed first (ie starting from index 0) followed by the extended addresses. The specification for the union, which holds a short or extended address, has already been described in MAC_addr_u

u8SDU is an array of **MAC_MAX_BEACON_PAYLOAD_LEN** bytes, which contains the beacon payload. The contents of the beacon payload are specified at the Application/NWK layer.

4.6.4 Poll Request

The *MLME-POLL.request* primitive is used to tell the MAC to attempt to retrieve pending data for the device from a coordinator in a non-beaconing PAN. The request is sent to the MAC using the **vAppApiMlmeRequest()** routine. The request structure is defined as follows:

```
struct tagMAC_MlmeReqPoll_s
{
    MAC_Addr_s sCoord;              /* Coordinator to poll for data */
    uint8      u8SecurityEnable;    /* True if security is to be used on
                                     * command frames
                                     */
} MAC_MlmeReqPoll_s;
```

sCoord contains the address of the coordinator to poll for data. The data structure in use has been described before in MAC_addr_s, and holds the PANid and either the 16-bit short address of the coordinator or its 64-bit extended address.

u8SecurityEnable if set to TRUE causes security processing to be applied to the data request frame which is sent to the coordinator. The coordinator address is used to look up the security information from the ACL in the PIB.

4.6.5 Poll Confirm

A Poll Confirm is generated by the MAC to inform the Application/NWK layer of the state of a Poll request The confirm message is sent to the Application/NWK layer using the callback routines registered at system start in the call **u32AppApiInit()**. It may also be sent synchronously to the Application/NWK layer as part of the **vAppApiMlmeRequest()** used to send the Poll Request. The structure of the Poll Confirm is as follows:

```
typedef struct
{
    uint8 u8Status; /* Status of data poll request */
} MAC_MlmeCfmPoll_s;
```

u8Status takes on a value from the **MAC_enum_e** enumeration type to indicate the status of the corresponding Poll request. The following values may be returned:

Value	Reason
MAC_ENUM_UNAVAILABLE_KEY	The security settings corresponding to the coordinator address are not found in the PIB ACL
MAC_ENUM_FAILED_SECURITY_CHECK	Security processing of the data request command fails for some other reason
MAC_ENUM_CHANNEL_ACCESS_FAILURE	The data request command cannot be sent due to the CSMA algorithm failing
MAC_ENUM_NO_ACK	No acknowledge frame is received for the data request command after the coordinator has tried to send the acknowledgement MAC_MAX_FRAME_RETRIES (3) times
MAC_ENUM_NO_DATA	No data is pending at the coordinator, or a data frame is not received within a timeout period after an acknowledge to the data request command is received, or a data frame with zero length payload is received
MAC_ENUM_INVALID_PARAMETER	A parameter in the Poll request is out of range or not supported
MAC_ENUM_SUCCESS	A data frame with non-zero payload length is received after the acknowledge of the data request command.

If the Poll confirm has status **MAC_ENUM_SUCCESS** to show that data is available, the data will be indicated to the Application/NWK layer using a *MCPS-DATA.indication* primitive.

4.6.6 Examples

The following is an example of a beacon synchronisation request.

```
/* Structures used to hold data for MLME request and response */
MAC_MlmeReqRsp_s   sMlmeReqRsp;
MAC_MlmeSyncCfm_s  sMlmeSyncCfm;

/* Create sync request on channel 11 */
sMlmeReqRsp.u8Type = MAC_MLME_REQ_SYNC;
sMlmeReqRsp.u8ParamLength = sizeof(MAC_MlmeReqSync_s);
sMlmeReqRsp.uParam.sReqSync.u8Channel = 11;
sMlmeReqRsp.uParam.sReqSync.u8TrackBeacon = TRUE;

/* Post sync request. There is no deferred confirm for this, we just
   get a SYNC-LOSS later if it didn't work */
vAppApiMlmeRequest(&sMlmeReqRsp, &sMlmeSyncCfm);
```

The following is an example of handling a beacon notify event (stores the beacon payload). The example assumes data is passed as a pointer to a deferred confirm indicator data type i.e. `MAC_MlmeDcfmInd_s *psMlmeInd.`

```
uint8 au8Payload[MAC_MAX_BEACON_PAYLOAD_LEN];
int i;

if (psMlmeInd->u8Type == MAC_MLME_IND_BEACON_NOTIFY)
{
    for (i = 0; i < psMlmeInd->uParam.sIndBeacon.u8SDUlength; i++))
    {
        /* Store beacon payload */
        au8Payload[i] = psMlmeInd->uParam.sIndBeacon.u8SDU[i];
    }
}
```

The following is an example of using a poll request to check if the coordinator has any data pending for the device. It is assumed that `u16CoordShortAddr` has been previously initialised.

```
#define DEMO_PAN_ID        0x1234

/* Structures used to hold data for MLME request and response */
MAC_MlmeReqRsp_s   sMlmeReqRsp;
MAC_MlmeSyncCfm_s  sMlmeSyncCfm;

/* Create a poll request */
sMlmeReqRsp.u8Type = MAC_MLME_REQ_POLL;
sMlmeReqRsp.u8ParamLength = sizeof(MAC_MlmeReqPoll_s);
sMlmeReqRsp.uParam.sReqPoll.u8SecurityEnable = FALSE;
sMlmeReqRsp.uParam.sReqPoll.sCoord.u8AddrMode = 2; /* Short address */
sMlmeReqRsp.uParam.sReqPoll.sCoord.u16PanId = DEMO_PAN_ID;
sMlmeReqRsp.uParam.sReqPoll.sCoord.uAddr.u16Short = u16CoordShortAddr;

/* Post poll request, response will be a deferred MLME-Poll.confirm.
   Will also receive a MCPS-Data.indication event if the coordinator has
   sent data. */
vAppApiMlmeRequest(&sMlmeReqRsp, &sMlmeSyncCfm);
```

4.7 Association

The MAC supports the Association feature as defined in ref [1] section 7.1.3 and 7.5.3

This feature allows a device to join a PAN and have pending data queued at the PAN coordinator. Before it can associate to a PAN however it must first find one. It should perform a *MLME-Reset.request* before performing either an Active or a Passive Scan using *MLME-Scan.request*, which will generate a list of PANs which have been found. The Application/NWK layer can then choose with which PAN it wishes to associate. At this point a *MLME-Associate.request* primitive is issued by the Application/NWK layer, which results in an Association request command being sent from the device to the coordinator. This frame command is acknowledged by the coordinator. After a time period has elapsed the device MAC sends a data request command to the coordinator to extract the result of the association. The coordinator acknowledges this command and is followed by an association response command from the coordinator which carries the status of the association attempt. On receiving the

association response the MAC generates a *MLME-ASSOCIATE.confirm* primitive giving the result of the association request.

At the coordinator, reception of the association request command results in the MLME raising a *MLME-ASSOCIATE.indication* to the Application/NWK layer which must process the indication and generate a *MLME-ASSOCIATE.response* primitive to the MAC. On receiving a data request from the device, this results in the association response command described above being sent to the device performing the association. The device will acknowledge reception of this command and the status of the *MLME-ASSOCIATE.response* will be reported to the coordinator by the MLME generating a *MLME-COMM-STATUS.indication*.

4.7.1 Associate Request

The *MLME-ASSOCIATE.request* primitive is used by the Application/NWK layer of an unassociated device to tell the MAC to attempt to request an association with a coordinator. The request is sent to the MAC using the **`MAC_vHandleMlmeReqRsp()`** routine. The request structure is defined as follows:

```
typedef struct
{
    MAC_Addr_s sCoord;           /* Coordinator to associate with */
    uint8      u8LogicalChan;    /* Logical channel to associate on */
    uint8      u8Capability;     /* Device's capability */
    uint8      u8SecurityEnable; /* True if security is to be used on
                                  * command frames
                                  */
} MAC_MlmeReqAssociate_s;
```

`sCoord` contains the address of the PAN coordinator to associate with. The data structure in use has been described before in MAC_addr_s, and holds the PANid and either the 16-bit short address of the coordinator or its 64-bit extended address.

`u8LogicalChan` contains the channel number (11 to 26 for the 2.45 GHz PHY) which the PAN to be associated with occupies

`u8Capability` is a byte encoded with the following information:

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

Alternate PAN coordinator – set to 1 if the device is capable of becoming a PAN coordinator.

Device Type – set to 1 if the device is an FFD, or 0 if an RFD.

Power Source – set to 1 if the device is mains powered, 0 otherwise.

Receiver on when idle – set to 1 if the device leaves its receiver on during idle periods and does not save power.

Security capability – set to 1 if the device can send and received frames using security.

Allocate address – set to 1 if the device requires the coordinator to provide a short address during the association procedure. If set to 0 the short address 0xFFFE is allocated in the

association response and the device will always communicate using the 64-bit extended address.

u8SecurityEnable is set to TRUE if security is to be used in this transfer.

4.7.2 Associate Confirm

An Associate Confirm is generated by the MAC to inform the Application/NWK layer of the state of an Association request. The confirm message is sent to the Application/NWK layer using the callback routines registered at system start in the call **MAC_vRegisterMlmeDcfmIndCallbacks()**. It may also be sent synchronously to the Application/NWK layer as part of the **MAC_vHandleMcpsReqRsp()** used to send the Associate Request. The structure of the Associate Confirm is as follows:

```
struct tagMAC_MlmeCfmAssociate_s
{
    uint8      u8Status;      /* Status of association */
    uint8      u8Pad;         /* Alignment */
    uint16     u16AssocShortAddr;
                              /* Associated Short Address */
} MAC_MlmeCfmAssociate_s;
```

u8Status holds the status of the operation, and takes on values from **MAC_Enum_e**

Value	Reason
MAC_ENUM_UNAVAILABLE_KEY	The security settings corresponding to the coordinator address were not found in the PIB ACL
MAC_ENUM_FAILED_SECURITY_CHECK	Security processing of the association request command fails for some other reason
MAC_ENUM_CHANNEL_ACCESS_FAILURE	The association request command cannot be sent due to the CSMA algorithm failing
MAC_ENUM_NO_ACK	No acknowledge frame is received for the association request command after the coordinator has tried to send the acknowledgement MAC_MAX_FRAME_RETRIES (3) times
MAC_ENUM_NO_DATA	No association response command was received within a timeout period after an acknowledge to the association request command is received
MAC_ENUM_INVALID_PARAMETER	A parameter in the Association request is out of range or not supported
0x01	PAN is full
0x02	Access to the PAN denied by the coordinator
MAC_ENUM_SUCCESS	The association request was successful

u16AssocShortAddr contains the short address allocated by the coordinator. If the address is 0xFFFE the device will use 64-bit extended addressing. If the association attempt failed it will hold the value 0xFFFF

4.7.3 Associate Indication

An Associate Indication is generated by the MAC to inform the Application/NWK layer that an association request command has been received. The indication message is sent to the Application/NWK layer using the callback routines registered at system start in the call **u32AppApiInit()**. The structure of the Associate Indication is as follows:

```
typedef struct
{
    MAC_ExtAddr_s sDeviceAddr;
                                /* Extended address of device wishing to
                                 * associate
                                 */
    uint8         u8Capability; /* Device capabilities */
    uint8         u8SecurityUse;/* True if security was used on command
                                 * frames
                                 */
    uint8         u8AclEntry;   /* Security suite used */
} MAC_MlmeIndAssociate_s;
```

sDeviceAddr contains the 64-bit extended address of the associating device

u8Capability holds the capabilities of the device as described in Associate Request

u8SecurityUse set to TRUE if the request command used security

u8AclEntry contains the security mode held in the ACL entry of the PIB for the device. If an ACL entry for the device cannot be found this value is set to 0x08. The security mode values are described in Scan confirm

4.7.4 Associate Response

An Associate Response is generated by the Application/NWK layer in response to receiving an Associate Indication. The response is sent using the **vAppApiMlmeRequest()** routine. It contains the following fields

```
struct tagMAC_MlmeRspAssociate_s
{
    MAC_ExtAddr_s sDeviceAddr;     /* Device's extended address */
    Uint16     u16AssocShortAddr;  /* Short address allocated to Device
                                    */
    uint8      u8Status;           /* Status of association */
    uint8      u8SecurityEnable;   /* True if security is to be used on
                                    * command frames
                                    */
} MAC_MlmeRspAssociate_s;
```

sDeviceAddr contains the associating device's 64-bit extended address

u16AssocShortAddr contains the 16-bit short address as allocated by the PAN coordinator. If the association was unsuccessful, the short address will be set to 0xFFFF

u8Status holds the result of the association request

Value	Description
0	Association successful
1	PAN is full
2	PAN access denied
3 - 0x7F	Reserved
0x80 – 0xFF	Reserved for MAC primitive enumeration values

`u8SecurityEnable` set to TRUE if security is being used on this transfer

4.7.5 Comm Status Indication

A Comm Status indication is issued by the MAC to the Application/NWK to report on the status of the Associate Response primitive. The format of the Comm Status indication has already been covered in 4.4.9 Comm Status Indication and so only the Status field values and the reasons for them will be described

Status	Reason
MAC_ENUM_UNAVAILABLE_KEY	Couldn't find a security key in the ACL for the transmission
MAC_ENUM_FAILED_SECURITY_CHECK	Failure during security processing of the frame
MAC_ENUM_TRANSACTION_OVERFLOW	No room available to store the association response command on the coordinator while waiting for data request from associating device
MAC_ENUM_TRANSACTION_EXPIRED	Association response was not retrieved by the associating device in the timeout period and has been discarded
MAC_ENUM_CHANNEL_ACCESS_FAILURE	Couldn't get access to the radio channel to perform the transmission
MAC_ENUM_NO_ACK	No acknowledgement from the associating device after sending the associate response command
MAC_ENUM_INVALID_PARAMETER	Invalid parameter value or parameter not supported in the Associate Response primitive
MAC_ENUM_SUCCESS	Associate response command sent successfully

4.7.6 Examples

The following is an example of a typical Associate request.

```
#define DEMO_PAN_ID                         0x1234
#define DEMO_COORD_ADDR                     0x0e00

/* Structures used to hold data for MLME request and response */
MAC_MlmeReqRsp_s  sMlmeReqRsp;
MAC_MlmeSyncCfm_s sMlmeSyncCfm;

/* Create associate request. We know short address and PAN ID of
   coordinator as this is preset and we have checked that received
   beacon matched this */
sMlmeReqRsp.u8Type = MAC_MLME_REQ_ASSOCIATE;
sMlmeReqRsp.u8ParamLength = sizeof(MAC_MlmeReqAssociate_s);
sMlmeReqRsp.uParam.sReqAssociate.u8LogicalChan = 11;
/* We want short address, other features off */
sMlmeReqRsp.uParam.sReqAssociate.u8Capability = 0x80;
sMlmeReqRsp.uParam.sReqAssociate.u8SecurityEnable = FALSE;
sMlmeReqRsp.uParam.sReqAssociate.sCoord.u8AddrMode = 2;
sMlmeReqRsp.uParam.sReqAssociate.sCoord.u16PanId = DEMO_PAN_ID;
sMlmeReqRsp.uParam.sReqAssociate.sCoord.uAddr.u16Short= DEMO_COORD_ADDR;

/* Put in associate request and check immediate confirm. Should be
   deferred, in which case response is handled by event handler */

vAppApiMlmeRequest(&sMlmeReqRsp, &sMlmeSyncCfm);

/* Handle synchronous confirm */
if (sMlmeSyncCfm.u8Status != MAC_MLME_CFM_DEFERRED)
{
    /* Unexpected result, expecting a deferred confirm */
}
```

The following is an example of a device handling an associate confirm event (it stores the short address assigned to it by the coordinator in variable `u16ShortAddr`). Assumes data is passed as a pointer to a deferred confirm indicator data type i.e. `MAC_MlmeDcfmInd_s *psMlmeInd`.

```
if (psMlmeInd->u8Type == MAC_MLME_DCFM_ASSOCIATE)
{
    if (psMlmeInd->uParam.sDcfmAssociate.u8Status == MAC_ENUM_SUCCESS)
    {
        /* Store short address */
        u16ShortAddr = psMlmeInd->
                       uParam.sDcfmAssociate.u16AssocShortAddr;
    }
}
```

The following is an example of a coordinator handling an Associate Indication message and generation of the appropriate response. Assumes data is passed as a pointer to a deferred confirm indicator data type i.e. `MAC_MlmeDcfmInd_s *psMlmeInd`.

```
/* Structures used to hold data for MLME request and response */
MAC_MlmeReqRsp_s  sMlmeReqRsp;
MAC_MlmeSyncCfm_s sMlmeSyncCfm;

tsDemoData sDemoData;

uint16 u16ShortAddress;
uint32 u32AddrLo;
uint32 u32AddrHi;
uint8 u8Node;
uint8 u8AssocStatus;

if (psMlmeInd->u8Type == MAC_MLME_IND_ASSOCIATE)
{
    /* Default short address */
    u16ShortAddress = 0xffff;

    /* Check node extended address matches and device wants short
       address */

    u32AddrLo = psMlmeInd->
                uParam.sIndAssociate.sDeviceAddr.u32L);
    u32AddrHi = psMlmeInd->
                uParam.sIndAssociate.sDeviceAddr.u32H);

    if ((u32AddrHi == DEMO_EXT_ADDR_HI)
       && (u32AddrLo >= DEMO_ENDPOINT_EXT_ADDR_LO_BASE)
       && (u32AddrLo < (DEMO_ENDPOINT_EXT_ADDR_LO_BASE
                            + DEMO_ENDPOINTS))
       && (psMlmeInd->uParam.sIndAssociate.u8Capability & 0x80))
    {
        /* Check if already associated (idiot proofing) */

        u8Node = 0;
        while (u8Node < sDemoData.sNode.u8AssociatedNodes)
        {
            if ((u32AddrHi ==
                 sDemoData.sNode.asAssocNodes[u8Node].u32ExtAddrHi)
                 && (u32AddrLo ==
                 sDemoData.sNode.asAssocNodes[u8Node].u32ExtAddrLo))
            {
                /*Already in system: give it same short address*/
                u16ShortAddress =
                sDemoData.sNode.asAssocNodes[u8Node].u16ShortAddr;
            }
            u8Node++;
        }

        /* Assume association succeeded */
        u8AssocStatus = 0;

        if (u16ShortAddress == 0xffff)
```

```
        {
            if (sDemoData.sNode.u8AssociatedNodes < DEMO_ENDPOINTS)
            {
                /*Allocate short address as next in list */
                u16ShortAddress = DEMO_ENDPOINT_ADDR_BASE
                + sDemoData.sNode.u8AssociatedNodes;
                /* Store details for future use */
                sDemoData.sNode.asAssocNodes
                [sDemoData.sNode.u8AssociatedNodes].u32ExtAddrHi
                = u32AddrHi;

                sDemoData.sNode.asAssocNodes
                [sDemoData.sNode.u8AssociatedNodes].u32ExtAddrLo
                = u32AddrLo;

                sDemoData.sNode.asAssocNodes
                [sDemoData.sNode.u8AssociatedNodes].u16ShortAddr
                = u16ShortAddress;
                sDemoData.sNode.u8AssociatedNodes++;
            }
            else
            {
                /* PAN access denied */
                u8AssocStatus = 2;
            }
        }
    }
    else
    {
        /* PAN access denied */
        u8AssocStatus = 2;
    }

    /* Create association response */
    sMlmeReqRsp.u8Type = MAC_MLME_RSP_ASSOCIATE;
    sMlmeReqRsp.u8ParamLength = sizeof(MAC_MlmeRspAssociate_s);
    memcpy(sMlmeReqRsp.uParam.sRspAssociate.sDeviceAddr,
           psMlmeInd->uParam.sIndAssociate.sDeviceAddr,
           MAC_EXT_ADDR_LEN);
    sMlmeReqRsp.uParam.sRspAssociate.u16AssocShortAddr =
           u16ShortAddress;
    sMlmeReqRsp.uParam.sRspAssociate.u8Status = u8AssocStatus;
    sMlmeReqRsp.uParam.sRspAssociate.u8SecurityEnable = FALSE;

    /* Send association response */
    vAppApiMlmeRequest(&sMlmeReqRsp, &sMlmeSyncCfm);

    /* There is no confirmation for an association response,
       hence no need to check */
```

4.8 Disassociation

The MAC supports the Disassociation feature as defined in ref [1] section 7.1.4 and 7.5.3

This feature allows a device which was previously associated with a PAN to stop being a member of that PAN. To disassociate from a PAN the device will issue a *MLME-DISASSOCIATE.request* primitive. It can also be used by a PAN coordinator to cause an associated device to leave the PAN.

The Application/NWK layer issues a Disassociate request. If this issued by a device a disassociation notification command is sent to the PAN coordinator. If the request was issued by a coordinator the notification command is stored for later transmission and the beacon contents are updated to show that there is a message pending for the device to be disassociated.

When a disassociation notification message has been transmitted an acknowledge is sent in return. On receiving the acknowledgement, the MAC generates a *MLME-DISASSOCIATE.confirm* to the Application/NWK layer.

If the Disassociation request was sent by a device, on receiving the disassociation notification command the coordinator MAC will generate a *MLME-DISASSOCIATE.indication* to indicate to the coordinator Application/Network layer that a device is leaving the PAN.

4.8.1 Disassociate Request

The *MLME-DISASSOCIATE.request* primitive is used by the Application/NWK layer of an associated device to tell the MAC to disassociate from the coordinator of a PAN. It is also used by the Application/NWK layer of a coordinator to remove an associated device from a PAN. The request is sent to the MAC using the **`MAC_vHandleMlmeReqRsp()`** routine. The request structure is defined as follows:

```
typedef struct
{
    MAC_Addr_s sAddr;       /* Disassociating address of other end */
    uint8 u8Reason;         /* Disassociation reason */
    uint8 u8SecurityEnable; /* True if security is to be used on command
                             * frames
                             */
} MAC_MlmeReqDisassociate_s;
```

`sAddr` contains the address of the recipient of the disassociation request – device or coordinator address (format described in 4.4.9 Comm Status Indication)

`u8Reason` holds the reason for the disassociation being requested:

Disassociation reason	Description
0	Reserved
1	Coordinator wishes device to leave the PAN
2	Device wishes to leave the PAN
0x03 – 0x7F	Reserved
0x80 – 0xFF	Reserved for MAC primitive enumeration values

u8SecurityEnable if set to TRUE indicates that security will be used during the transactions

4.8.2 Disassociate Confirm

An Disassociate Confirm is generated by the MAC to inform the Application/NWK layer of the state of a Disassociate Request. The confirm message is sent to the Application/NWK layer using the callback routines registered at system start in the call **u32AppApiInit()**. It may also be sent synchronously to the Application/NWK layer as part of the **vAppApiMlmeRequest()** used to send the Disassociate Request. The structure of the Disassociate Confirm is as follows:

```
typedef struct
{
    uint8 u8Status; /* Status of disassociation */
} MAC_MlmeCfmDisassociate_s;
```

u8Status contains the result of the corresponding Disassociate Request:

Status	Reason
MAC_ENUM_UNAVAILABLE_KEY	Couldn't find a security key in the ACL for the transmission
MAC_ENUM_FAILED_SECURITY_CHECK	Failure during security processing of the frame
MAC_ENUM_TRANSACTION_OVERFLOW	No room available to store the disassociation notification command on the coordinator - when coordinator requests disassociation
MAC_ENUM_TRANSACTION_EXPIRED	Disassociation notification command was not retrieved by the intended device in the timeout period and has been discarded (coordinator requested disassociation)
MAC_ENUM_CHANNEL_ACCESS_FAILURE	Couldn't get access to the radio channel to perform the transmission of the disassociate notification command
MAC_ENUM_NO_ACK	No acknowledgement from the associating device after sending the disassociate notification command
MAC_ENUM_INVALID_PARAMETER	Invalid parameter value or parameter not supported in the Disassociate Request primitive
MAC_ENUM_SUCCESS	Disassociate request completed successfully

4.8.3 Disassociate Indication

A Disassociate Indication is generated by the MAC to inform the Application/NWK layer that a disassociate notification command has been received. The indication message is sent to the Application/NWK layer using the callback routines registered at system start in the call **u32AppApiInit()**. The structure of the Disassociate Indication is as follows:

```
typedef struct
{
    MAC_ExtAddr_s sDeviceAddr; /* Extended address of device which has
                                * sent disassociation notification
                                */
    uint8      u8Reason;       /* Reason for disassociating */
    uint8      u8SecurityUse;  /* True if security was used on command
                                * frames
                                */
    uint8      u8AclEntry;     /* Security suite used */
} MAC_MlmeIndDisassociate_s;
```

sDeviceAddr contains the 64-bit extended address of the device, which generated the disassociation request

u8Reason contains the reason for the disassociation as described in 4.8.1 Disassociate Request

u8SecurityUse TRUE if security is being used during the transmission

u8AclEntry contains the security mode held in the ACL entry of the PIB for the device. If an ACL entry for the device cannot be found this value is set to 0x08. The security mode values are described in Scan confirm

4.8.4 Examples

The following is an example of a request to disassociate a device from a PAN

```
/* Structures used to hold data for MLME request and response */
MAC_MlmeReqRsp_s   sMlmeReqRsp;
MAC_MlmeSyncCfm_s  sMlmeSyncCfm;

/* Post disassociate request for device to leave PAN */
sMlmeReqRsp.u8Type = MAC_MLME_REQ_DISASSOCIATE;
sMlmeReqRsp.u8ParamLength = sizeof(MAC_MlmeReqDisassociate_s);
sMlmeReqRsp.uParam.sReqDisassociate.sAddr.u8AddrMode = 2; /* Short */
sMlmeReqRsp.uParam.sReqDisassociate.sAddr.uAddr.u16Short =
                                                    u16CoordShortAddr;
sMlmeReqRsp.uParam.sReqDisassociate.u8Reason = 2; /* Device leave PAN */
sMlmeReqRsp.uParam.sReqDisassociate.u8SecurityEnable = FALSE;

vAppApiMlmeRequest(&sMlmeReqRsp, &sMlmeSyncCfm);

/* Handle synchronous confirm */
if (sMlmeSyncCfm.u8Status != MAC_MLME_CFM_DEFERRED)
{
    /* Unexpected result, expecting a deferred confirm */
}
```

4.9 Data transmission and reception

The MAC provides a data service for the transmission and reception of data. Data is transmitted using the *MCPS-DATA.request*; the status of the transmission is reported by the *MCPS-DATA.confirm*. Reception of data is indicated to the Application/NWK layer by the MAC raising a *MCPS-DATA.indication*.

4.9.1 Data Request

The *MCPS-DATA.request* primitive is used by the Application/NWK layer to transmit a frame of data to a destination device. The request is sent to the MAC using the **`vAppApiMcpsRequest()`** routine. The request structure is defined as follows:

```
struct tagMAC_McpsReqData_s
{
    uint8             u8Handle; /* Handle of frame in queue */
    MAC_TxFrameData_s sFrame;   /* Frame to send */
} MAC_McpsReqData_s;
```

`u8Handle` identifies the transmission allowing more than one transmission to be performed before the corresponding confirm has been seen. It may take the values 0 to 0xFF; the handle is generated by the Application/NWK layer.

`sFrame` contains the data frame to be sent by this request and has the following format:

```
typedef struct
{
    MAC_Addr_s     sSrcAddr;          /* Source address */
    MAC_Addr_s     sDstAddr;          /* Destination address */
    uint8          u8TxOptions;       /* Transmit options */
    uint8          u8SduLength;       /* Length of payload (MSDU) */
    uint8          au8Sdu[MAC_MAX_DATA_PAYLOAD_LEN];
                                      /* Payload (MSDU) */
} MAC_TxFrameData_s;
```

`sSrcAddr` describes the source address for the transmission.

`sDstAddr` describes the destination address for the transmission.

Both **`sSrcAddr`** and **`sDstAddr`** are of type MAC_Addr_s which is described in MAC_addr_s; this structure allows an address to be specified either as a 16-bit short address or as a 64-bit extended address. It also allows the PAN identifier for each address to be included.

`u8TxOptions` contains the options for this transmission, encoded as follows

Bits 7 - 4	Bit 3	Bit 2	Bit 1	Bit 0
0000	Security Enabled transmission	Indirect Transmission	GTS Transmission	Acknowledged Transmission

The above bits are set to 1 to invoke the option. A GTS Transmission overrides an Indirect Transmission option. The indirect transmission option is only valid for a coordinator generated data request; for a non-coordinator device the option is ignored. If the Security option is set the ACL corresponding to the destination address is searched and the keys etc used to apply security to the data frame to be sent.

u8SduLength denotes the length of the payload field of the message in bytes

au8Sdu is the array of bytes making up the payload of the transmission, up to **MAC_MAX_DATA_PAYLOAD_LEN** (118) in length depending on overhead from the frame header.

4.9.2 Data Confirm

An *MCPS-DATA.confirm* primitive is generated by the MAC to inform the Application/NWK layer of the state of a *MCPS-DATA.request*. The confirm message is sent to the Application/NWK layer using the callback routines registered at system start in the call **u32AppApiInit()**. It may also be sent synchronously to the Application/NWK layer as part of the **vAppApiMcpsRequest()**call used to send the Data Request. The structure of the Data Confirm is as follows:

```
typedef struct
{
    uint8 u8Handle; /* Handle matching associated request */
    uint8 u8Status; /* Status of request */
} MAC_McpsCfmData_s;
```

u8Handle contains the handle of the *MCPS-DATA.request* whose status is being reported

u8Status carries the result of the *MCPS-DATA.request*. It may take the following values

Status	Reason
MAC_ENUM_UNAVAILABLE_KEY	Couldn't find a security key in the ACL for the transmission
MAC_ENUM_FAILED_SECURITY_CHECK	Failure during security processing of the frame
MAC_ENUM_FRAME_TOO_LONG	The size of the frame after security processing is greater than the maximum size that can be transmitted, or the transmission is too long to fit in the CAP or GTS period
MAC_ENUM_INVALID_GTS	No Guaranteed Time Slot allocated for this destination
MAC_ENUM_TRANSACTION_OVERFLOW	No room available to store the data when an indirect transmission is specified in the Tx Options when a coordinator requests the transmission
MAC_ENUM_TRANSACTION_EXPIRED	Disassociation notification command was not retrieved by the intended device in the timeout period and has been discarded (coordinator requested disassociation)
MAC_ENUM_CHANNEL_ACCESS_FAILURE	Couldn't get access to the radio channel to perform the transmission of the data frame
MAC_ENUM_NO_ACK	No acknowledgement from the destination device after sending the data frame with the acknowledge option set
MAC_ENUM_INVALID_PARAMETER	Invalid parameter value or parameter not supported in the Data Request primitive

Status	Reason
	supported in the Data Request primitive
MAC_ENUM_SUCCESS	Data request completed successfully

4.9.3 Data Indication

An *MCPS-DATA.indication* is generated by the MAC to inform the Application/NWK layer of the reception of a data packet. The indication message is sent to the Application/NWK layer using the callback routines registered at system start in the call **`vAppApiMcpsRequest()`**. The structure of the Data Indication is as follows:

```
typedef struct
{
    MAC_RxFrameData_s sFrame;   /* Frame received */
} MAC_McpsIndData_s;
```

`sFrame` is made up of the following type:

```
struct tagMAC_RxFrameData_s
{
    MAC_Addr_s     sSrcAddr;      /* Source address */
    MAC_Addr_s     sDstAddr;      /* Destination address */
    uint8          u8LinkQuality; /* Link quality of received frame */
    uint8          u8SecurityUse; /* True if security was used */
    uint8          u8AclEntry;    /* Security suite used */
    uint8          u8SduLength;   /* Length of payload (MSDU) */
    uint8          au8Sdu[MAC_MAX_DATA_PAYLOAD_LEN];
                                  /* Payload (MSDU) */
} MAC_RxFrameData_s;
```

`sSrcAddr` holds the source address and **`sDstAddr`** holds the and destination addresses – both of which may be short (16-bit) or extended (64-bit) format together with the PAN identifier of each address. The details of this structure are described in 4.9.1 Data Request.

`u8LinkQuality` contains a value between 0 and 0xFF which gives the quality of the reception of the received frame.

`u8SecurityUse` indicates if security was used in transmitting the data.

`u8AclEntry` indicates the security suite used during the transmission, as retrieved from the ACL for the source address held in the PIB. The encoding of this field is given in Security_modes.

`u8SduLength` contains the length of the payload in bytes.

`au8Sdu` is the array of bytes containing the payload of the transmission.

4.9.4 Purge Request

The *MCPS-PURGE.request* primitive is used by the Application/NWK layer to remove a data frame from a transaction queue where it is held prior to transmission. The request is sent to the MAC using the **vAppApiMcpsRequest()** routine. The request structure is defined as follows:

```
typedef struct
{
    uint8          u8Handle;    /* Handle of request to purge */
} MAC_McpsReqPurge_s;
```

u8Handle contains the handle of the Data Request to be removed from the transaction queue

4.9.5 Purge Confirm

An *MCPS-PURGE.confirm* primitive is generated by the MAC to inform the Application/NWK layer of the result of a *MCPS-PURGE.request* primitive. The confirm message is sent to the Application/NWK layer using the callback routines registered at system start in the call **u32AppApiInit()**. It may also be sent synchronously to the Application/NWK layer as part of the **vAppApiMcpsRequest()** used to send the Purge Request. The structure of the Purge Confirm is as follows:

```
typedef struct
{
    uint8 u8Handle; /* Handle matching associated request */
    uint8 u8Status; /* Status of request (uses MAC_Enum_e) */
} MAC_McpsCfmPurge_s;
```

u8Handle holds the handle of the transaction specified in the Purge Request

u8Status contains the result of the attempt to remove the data from the transaction queue. It takes on values from the enumeration **MAC_enum_e**.

Status	Reason
MAC_ENUM_INVALID_HANDLE	Could not find a transaction with a handle matching that of the purge request
MAC_ENUM_SUCCESS	Purge request completed successfully

4.9.6 Examples

The following is an example of a device transmitting data to a coordinator using a data request. The variable `u8CurrentTxHandle` is set at a higher layer and is just used as a data frame tag. The variable `u16ShortAddr` contains the short address of the device that is transmitting the data.

```
#define DEMO_PAN_ID                         0x0e1c
#define DEMO_COORD_ADDR                     0x0e00

/* Structures used to hold data for MLME request and response */
MAC_McpsReqRsp_s  sMcpsReqRsp;
MAC_McpsSyncCfm_s sMcpsSyncCfm;

uint8 *pu8Payload;

/* Create frame transmission request */
sMcpsReqRsp.u8Type = MAC_MCPS_REQ_DATA;
sMcpsReqRsp.u8ParamLength = sizeof(MAC_McpsReqData_s);

/* Set handle so we can match confirmation to request */
sMcpsReqRsp.uParam.sReqData.u8Handle = u8CurrentTxHandle;

/* Use short address for source */
sMcpsReqRsp.uParam.sReqData.sFrame.sAddr.sSrc.u8AddrMode = 2;
sMcpsReqRsp.uParam.sReqData.sFrame.sAddr.sSrc.u16PanId = DEMO_PAN_ID;
sMcpsReqRsp.uParam.sReqData.sFrame.sAddr.sSrc.uAddr.u16Short =
                                                        u16ShortAddr;

/* Use short address for destination */
sMcpsReqRsp.uParam.sReqData.sFrame.sAddr.sDst.u8AddrMode = 2;
sMcpsReqRsp.uParam.sReqData.sFrame.sAddr.sDst.u16PanId = DEMO_PAN_ID;
sMcpsReqRsp.uParam.sReqData.sFrame.sAddr.sDst.uAddr.u16Short =
                                                        DEMO_COORD_ADDR;

/* Frame requires ack but not security, indirect transmit or GTS */
sMcpsReqRsp.uParam.sReqData.sFrame.u8TxOptions = MAC_TX_OPTION_ACK;

/* Set payload, only use first 8 bytes */
sMcpsReqRsp.uParam.sReqData.sFrame.u8SduLength = 8;
pu8Payload = sMcpsReqRsp.uParam.sReqData.sFrame.au8Sdu;
pu8Payload[0] = 0x00;
pu8Payload[1] = 0x01;
pu8Payload[2] = 0x02;
pu8Payload[3] = 0x03;
pu8Payload[4] = 0x04;
pu8Payload[5] = 0x05;
pu8Payload[6] = 0x06;
pu8Payload[6] = 0x07;

/* Request transmit */
vAppApiMcpsRequest(&sMcpsReqRsp, &sMcpsSyncCfm);
```

A Data Confirm can be sent to the application via callbacks

```
PRIVATE void vProcessIncomingMcps(MAC_McpsDcfmInd_s *psMcpsInd)
{
    /* Process MCPS indication by checking if it is a confirmation of
       our outgoing frame */
    if ((psMcpsInd->u8Type == MAC_MCPS_DCFM_DATA)
        && (sDemoData.sSystem.eState == E_STATE_TX_DATA))
    {
        if (psMcpsInd->uParam.sDcfmData.u8Handle ==
                                sDemoData.sTransceiver.u8CurrentTxHandle)
        {
            /* Increment handle for next time. Increment failures */
            sDemoData.sTransceiver.u8CurrentTxHandle++;

            /* Start to read sensors. This takes a while but rather than
            wait for an interrupt we just poll and, once finished, move
            back to the running state to wait for the next beacon. Not a
            power saving solution! */
            sDemoData.sSystem.eState = E_STATE_READ_SENSORS;
            vProcessRead();
            sDemoData.sSystem.eState = E_STATE_RUNNING;
        }
    }
}
```

The following is an example of handling the data indication event that is generated by the MAC layer of a coordinator when data is received. The variable `u16DeviceAddr` contains the short address of the device from which we want to receive data. Assumes data is passed as a pointer to a deferred confirm indicator data type i.e. `MAC_McpsDcfmInd_s *psMcpsInd`.

```
MAC_RxFrameData_s *psFrame;
MAC_Addr_s *psAddr;
uint16 u16NodeAddr;
au8DeviceData[8];

if (psMcpsInd->u8Type == MAC_MCPS_IND_DATA)
{
    psFrame = &psMcpsInd->uParam.sIndData.sFrame;
    psAddr = &psFrame->sAddrPair.sSrc;

    /* Using short addressing mode */
    if (psAddr->u8AddrMode == 2)
    {
        /* Get address of device that is sending the data */
        u16NodeAddr = psAddr->uAddr.u16Short;
        /* If this is the device we want */
        if (u16NodeAddr == u16DeviceAddr)
        {
            /* Store the received data, only interested in 8 bytes */
            for(i = 0; i < 8; i++)
            {
                au8DeviceData[i] = psFrame->au8Sdu[i];
            }
        }
    }
}
```

The following is an example of a request to purge a data frame from the transaction queue. The variable `u8PurgeItemHandle` defines which item is to be purged and is set by a higher layer.

```
/* Structures used to hold data for MLME request and response */
MAC_McpsReqRsp_s  sMcpsReqRsp;
MAC_McpsSyncCfm_s sMcpsSyncCfm;

/* Send request to remove a data frame from transaction queue */
sMcpsReqRsp.u8Type = MAC_MCPS_REQ_PURGE;
sMlmeReqRsp.u8ParamLength = sizeof(MAC_McpsReqPurge_s);
sMlmeReqRsp.uParam.sReqPurge.u8Handle = u8PurgeItemHandle;

/* Request transmit */
vAppApiMcpsRequest(&sMcpsReqRsp, &sMcpsSyncCfm);
```

The following is an example of handling a purge confirm event. Assumes data is passed as a pointer to a deferred confirm indicator data type i.e. `MAC_McpsDcfmInd_s *psMcpsInd.`

```
if (psMcpsInd->u8Type == MAC_MCPS_DCFM_PURGE)
{
    if (psMcpsInd->uParam.sCfmPurge.u8Status != MAC_ENUM_SUCCESS)
    {
        /* Purge request failed */
    }
}
```

4.10 Rx Enable

The MAC supports the Receiver Enable feature as defined in ref [1] section 7.1.10

This feature allows a device to control when its receiver will be enabled or disabled, and for how long. On beacon-enabled PANs the timings are relative to superframe boundaries; on non-beacon-enabled PANs the receiver is enabled immediately.

4.10.1 Rx Enable Request

The *MLME-RX-ENABLE.request* primitive is used by the Application/NWK layer to request that the receiver is turned at a particular time and for how long. The request is sent to the MAC using the **vAppApiMlmeRequest()** routine. The request structure is defined as follows:

```
struct tagMAC_MlmeReqRxEnable_s
{
    uint32 u32RxOnTime;         /* Number of symbol periods from the
                                 * start of the superframe before the
                                 * receiver is enabled (beacon networks
                                 * only)
                                 */
    uint32 u32RxOnDuration;     /* Number of symbol periods the receiver
                                 * should be enabled for
                                 */
    uint8  u8DeferPermit;       /* True if receiver enable can be
                                 * deferred to the next superframe
                                 * (beacon networks only)
                                 */
} MAC_MlmeReqRxEnable_s;
```

u32RxOnTime is a 32-bit quantity specifying the number of symbols after the start of the superframe that the receiver should be enabled

u32RxOnDuration is a 32-bit quantity specifying the number of symbols that the receiver should remain enabled. If equal to 0, the receiver is disabled.

u8DeferPermit set to TRUE if the enable period is to be allowed to start in the next full superframe period if the requested on time has already passed in the current superframe.

A new Rx Enable Request must be generated for each attempt to enable the receiver

4.10.2 Rx Enable Confirm

An MLME-RX-ENABLE.confirm primitive is generated by the MAC to inform the Application/NWK layer of the result of a *MLME-RX-ENABLE.request* primitive. The confirm message is sent to the Application/NWK layer using the callback routines registered at system start in the call **u32AppApiInit()**. It may also be sent synchronously to the Application/NWK layer as part of the **vAppApiMlmeRequest()** used to send the Rx Enable Request. The structure of the Rx Enable Confirm is as follows:

```
typedef struct
{
    uint8 u8Status; /* Status of receiver enable request */
} MAC_MlmeCfmRxEnable_s;
```

u8Status contains the result of the Rx Enable Request, taking on values from MAC_enum_e

Status	Reason
MAC_ENUM_INVALID_PARAMETER	The combination of start time and duration requested will not fit inside the superframe (only relevant for a beacon enabled PAN)
MAC_ENUM_OUT_OF_CAP	The start time requested has passed and the receive is not allowed to be deferred to the next superframe period or the requested duration will not fit in the current CAP (only relevant for a beacon enabled PAN)
MAC_ENUM_TX_ACTIVE	The receiver cannot be enabled because the transmitter is active
MAC_ENUM_SUCCESS	Rx Enable request completed successfully

4.10.3 Examples

The following is an example of an receiver enable request.

```
#define RX_ON_TIME            0x00
#define RX_ON_DURATION        0x200000

/* Structures used to hold data for MLME request and response */
MAC_MlmeReqRsp_s   sMlmeReqRsp;
MAC_MlmeSyncCfm_s  sMlmeSyncCfm;

/* Post receiver enable request */
sMlmeReqRsp.u8Type = MAC_MLME_REQ_RX_ENABLE;
sMlmeReqRsp.u8ParamLength = sizeof(MAC_MlmeReqRxEnable_s);
sMlmeReqRsp.uParam.sReqRxEnable.u8DeferPermit = TRUE;
sMlmeReqRsp.uParam.sReqRxEnable.u32RxOnTime = RX_ON_TIME;
sMlmeReqRsp.uParam.sReqRxEnable.u32RxOnDuration = RX_ON_DURATION);
vAppApiMlmeRequest(&sMlmeReqRsp, &sMlmeSyncCfm);

/* Handle response */
if (sMlmeSyncCfm.u8Status != MAC_ENUM_SUCCESS)
{
    /* Receiver not enabled */
}
```

4.11 Guaranteed Time Slots (GTS)

The MAC supports the Guaranteed Time Slot feature as defined in ref [1] sections 7.1.7 and 7.5.7

Guaranteed time slots allow portions of a superframe to be assigned to a device for its exclusive use, to allow communications between the device and PAN coordinator. Up to 7 GTS slots can be allocated provided there is enough room in the superframe; a slot may be multiple superframe slots in length. The PAN coordinator is responsible for allocating and deallocating GTSs. Requests for allocation of GTSs are made by devices; GTSs may be deallocated by the PAN coordinator or by the device, which owns a slot giving it up. A GTS has a defined direction

(transmit or receive relative to the device) and a device may request a transmit GTS and a receive GTS. A device must be tracking beacons in order to be allowed to use GTSs.

The result of an allocation or deallocation of a GTS is transmitted in the beacon; in the case of the allocation, information such as the start slot, slot length and the device short address are transmitted as part of the GTS descriptor. The contents of the beacon are examined to allow the GTS Confirm primitive to report the status of the GTS Request

4.11.1 GTS Request

The *MLME-GTS.request* primitive is used by the Application/NWK layer to request that the receiver is turned at a particular time and for how long. The request is sent to the MAC using the **vAppApiMlmeRequest()** routine. The request structure is defined as follows:

```
typedef struct
{
    uint8  u8Characteristics;   /* GTS characteristics */
    uint8  u8SecurityEnable;    /* True if security is to be used on
                                 * command frames
                                 */
} MAC_MlmeReqGts_s;
```

u8Characteristics contains the characteristics of the GTS being requested, encoded in a byte as shown below

Bits 0 - 3	Bit 4	Bit 5	Bits 6 – 7
GTS length (in superframe slots)	GTS direction (0 = Transmit 1 = Receive)	Characteristics type (1 = GTS allocation 0 = GTS deallocation)	Reserved

GTS direction is defined relative to the device

u8SecurityEnable is set to TRUE if security is to be used during the request

4.11.2 GTS Confirm

An *MLME-GTS.confirm* primitive is generated by the MAC to inform the Application/NWK layer of the result of a *MLME-GTS.request* primitive. The confirm message is sent to the Application/NWK layer using the callback routines registered at system start in the call **u32AppApiInit()**. It may also be sent synchronously to the Application/NWK layer as part of the **vAppApiMlmeRequest()** used to send the GTS Request. The structure of the GTS Confirm is as follows:

```
typedef struct
{
    uint8 u8Status;             /* Status of GTS request */
    uint8 u8Characteristics;    /* GTS characteristics */
} MAC_MlmeCfmGts_s;
```

u8Status contains the result of the GTS request using the MAC_enum_e enumeration.

u8Characteristics carries the characteristics of the GTS that has been allocated as encoded in 4.11.1 GTS Request. If a GTS has been deallocated the characteristics type field is set to 0.

Status	Reason
MAC_ENUM_NO_SHORT_ADDRESS	Generated if the requesting device has a short address of 0xFFFE or 0xFFFF
MAC_ENUM_UNAVAILABLE_KEY	Couldn't find a security key in the ACL for the transmission (only if security in use)
MAC_ENUM_FAILED_SECURITY_CHECK	Failure during security processing of the frame (only if security in use)
MAC_ENUM_CHANNEL_ACCESS_FAILURE	Couldn't get access to the radio channel to perform the transmission of the GTS request frame
MAC_ENUM_NO_ACK	No acknowledgement from the destination device after sending the GTS request frame
MAC_ENUM_NO_DATA	A beacon containing a GTS descriptor corresponding to the device short address was not received within the required time, or a MLME-SYNC-LOSS.indication primitive was received with a MAC_ENUM_BEACON_LOSS status
MAC_ENUM_DENIED	The GTS allocation request has been denied by the PAN coordinator
MAC_ENUM_INVALID_PARAMETER	Invalid parameter value or parameter not supported in the GTS Request primitive
MAC_ENUM_SUCCESS	GTS successfully allocated or deallocated

4.11.3 GTS Indication

A GTS Indication is generated by the MAC to inform the Application/NWK layer that a GTS request command to allocate or deallocate a GTS has been received, or on a PAN coordinator where the GTS deallocation is generated by the coordinator itself. The indication message is sent to the Application/NWK layer using the callback routines registered at system start in the call **`u32AppApiInit()`**. The structure of the GTS Indication is as follows:

```
typedef struct
{
    uint16 u16ShortAddr;
                                /* Short address of device to which GTS
                                 * has been allocated or deallocated
                                 */
    uint8 u8Characteristics;    /* Characteristics of the GTS */
    uint8 u8Security;           /* True if security was used on command
                                 * frames
                                 */
    uint8 u8AclEntry;           /* Security suite used */
} MAC_MlmeIndGts_s;
```

`u16ShortAddr` contains the 16-bit short address of the device to which the GTS has been allocated or deallocated, with value between 0 and 0xFFFD

u8Characteristics carries the characteristics of the GTS that has been allocated as encoded in 4.11.1 GTS Request. If a GTS has been deallocated the characteristics type field is set to 0

u8Security is set to TRUE if security is used in the transmission of frames between the device and coordinator

u8AclEntry holds the value of the security mode from the ACL entry associated with the sender of the GTS request command, ie the security mode used in the transmission

4.11.4 Examples

The following is an example of a device making a GTS request to a PAN co-ordinator:

```
/* Structures used to hold data for MLME request and response */
MAC_MlmeReqRsp_s  sMlmeReqRsp;
MAC_MlmeSyncCfm_s sMlmeSyncCfm;

uint8 u8Characteristics = 0;

/* Make GTS request for 4 slots, in tx direction */
sMlmeReqRsp.u8Type = MAC_MLME_REQ_GTS;
sMlmeReqRsp.u8ParamLength = sizeof(MAC_MlmeReqGts_s);
sMlmeReqRsp.uParam.MAC_MlmeReqGts_s.u8SecurityEnable = TRUE;

/* characterstics defined in mac_sap.h */
u8Characteristics |= 4 << MAC_GTS_LENGTH_BIT;
u8Characteristics |= MAC_GTS_DIRECTION_TX << MAC_GTS_DIRECTION_BIT;
u8Characteristics |= MAC_GTS_TYPE_ALLOC << MAC_GTS_TYPE_BIT;

sMlmeReqRsp.uParam.MAC_MlmeReqGts_s.u8Characteristics =
                                                    u8Characteristics;

/* Put in associate request and check immediate confirm. Should
   be deferred, in which case response is handled by event handler */
vAppApiMlmeRequest(&sMlmeReqRsp, &sMlmeSyncCfm);

/* Handle synchronous confirm */
if (sMlmeSyncCfm.u8Status != MAC_MLME_CFM_DEFERRED)
{
    /* Unexpected result - handle error*/
}
```

The following is an example of handling a deferred GTS confirm (generated by the MAC layer in response to the above request). Assumes data is passed as a pointer to a deferred confirm indicator data type i.e. `MAC_MlmeDcfmInd_s *psMlmeInd.`

```
if (psMlmeInd->u8Type == MAC_MLME_DCFM_GTS)
{
    if (psMlmeInd->uParam.MAC_MlmeCfmGts_s.u8Status == MAC_ENUM_SUCCESS)
    {
        /* GTS allocated successfully, store characteristics */
        u8Characteristics = psMlmeInd->
                            uParam.MAC_MlmeCfmGts_s.u8Characteristics;
        u8GtsLength = (u8Characteristics & MAC_GTS_LENGTH_MASK);
        u8GtsDirection = (u8Characteristics & MAC_GTS_DIRECTION_MASK) >>
                                                    MAC_GTS_DIRECTION_BIT;
        u8GtsType = (u8Characteristics & MAC_GTS_TYPE_MASK) >>
                                                        MAC_GTS_TYPE_BIT;
    }
}
```

The following example shows a coordinator handling a GTS indication event (generated by the MAC layer whenever following reception of a GTS request command from a device). Assumes data is passed as a pointer to a deferred confirm indicator data type i.e. `MAC_MlmeDcfmInd_s *psMlmeInd.`

```
if (psMlmeInd->u8Type == MAC_MLME_IND_GTS)
{
    /* determine whether allocation or de-allocation has occurred */
    u8Characteristics = psMlmeInd->
                                uParam.MAC_MlmeIndGts_s.u8Characteristics;
    u8GtsType = (u8Characteristics & MAC_GTS_TYPE_MASK) >>
                                                        MAC_GTS_TYPE_BIT;

    if (u8GtsType == MAC_GTS_TYPE_DEALLOC)
    {
    /* handle de-allocation of GTS */
    }
    else
    {
    /* handle allocation of GTS */
    }
}
```

Appendix A

Identifying modules

All modules mounted on Sensor and Controller boards found in Evaluation, Starter or Expansion kits have a barcode label easily visible, stuck over the JN5121/JN513x or shielding can. The labels on modules with a screening can have a 10-digit batch and serial number below the barcode. The first 4 digits are the batch code. Labels on modules without a screening can have a 4-digit batch code above the barcode.

The batch code shows the week of manufacture of the module. It ends in either 05 or 06 for the year of manufacture; the first two digits represent the week number starting from January 1st (week 01) in the year. So batch code 1006 would be week 10 2006. The batch code shows which version of the JN5121 or JN513x is used on the module. For a module with a screening can a typical batch and serial number would be 0906300180, showing it was built in week 9 of 2006

Modules with batch codes later than 1006 contain the MAC in ROM, those with earlier codes require the MAC library to be built into the program and downloaded to the flash memory.

Identifying packaged devices

Devices supplied directly (i.e. NOT mounted on modules) can be identified by a 4-digit date code. Chip labelling is in the following format:

First line	- Jennic
Second line	- Chip name (e.g. JN 5121)
Third line	- ABN number
Fourth line	- Date code

The date code is in the form YYWW (year number followed by week number). Devices with date codes of 0607 or later have the MAC in ROM. Those with earlier codes require the MAC library to be built into the program and downloaded to the Flash memory.

References

[1] IEEE Std. 802.15.4-2003

[2] Jennic Integrated Peripherals API Reference Manual (JN-RM-2001)

Jennic is a fabless semiconductor company leading the wireless connectivity revolution into new applications. Jennic combines expertise in systems and software with world class RF and digital chip design to provide low cost, highly integrated silicon solutions for its customers and partners. Headquartered in Sheffield, UK, Jennic is privately held and has a proven track record of successful silicon chip development.

Jennic

TECHNOLOGY FOR A CHANGING WORLD

Jennic Ltd.
Furnival Street, Sheffield, S1 4QT, UK
Tel: +44 (0) 114 281 2655 Fax: +44 (0) 114 281 2951
Email: info@jennic.com Web: www.jennic.com