# G3 PHYSICAL LAYER API SPECIFICATION

**Document Revision:** 0.6

**Issue Date:** 08 August, 2011

## Revision History

| Revision | Draft | Author | Date | Comment |
|---|---|---|---|---|
| 0.0 | Initial version | Susan Yim | 01/21/2010 | |
| 0.1 | PHY Lib release 1.0 | Susan Yim<br>Minghua Fu<br>Gary Xu | 06/09/2010 | |
| 0.2 | PHY Lib release 1.1 | Minghua Fu<br>Gary Xu | 07/31/2010 | Added tone map feature |
| 0.3 | PHY Lib release 1.2 | Minghua Fu<br>Gary Xu | 09/15/2010 | Added sub band SNR, LQI and updated tone map definition |
| 0.4 | PHY Lib release 1.3 | Minghua Fu<br>Gary Xu | 02/24/2011 | Updated tone mask definition |
| 0.5 | PHY Lib release 1.4 | Gary Xu | 03/18/2011 | Removed HAL API and updated tone mask definition |
| 0.6 | PHY Lib release 2.1 | Gary Xu | 08/08/2011 | Added TX preparation API functions |

## Sign Off List

| Group | Title | Authority | Date | Comment |
|---|---|---|---|---|
| | | | | |

# Table of Content


1.0 Introduction ..... 5
2.0 G3 PHY Layer Public APIs ..... 6
2.1 PHY_getLibVersion ..... 8
2.2 PHY_init ..... 8
2.3 PHY Tx Path Initialization ..... 9
2.4 PHY Transmit Prepare PPDU ..... 9
2.5 PHY Transmit PPDU ..... 11
2.6 PHY Transmit Prepare ACK Frame ..... 12
2.7 PHY Transmit ACK Frame ..... 13
2.8 PHY TX Process ..... 13
2.9 PHY TX Parameter Set ..... 14
2.10 PHY TX Parameter Get ..... 15
2.11 PHY TX Get Statistics ..... 16
2.12 PHY Layer Rx Path Initialization ..... 16
2.13 PHY Receiver Start Channel Acquisition ..... 17
2.14 PHY Receiver Suspend ..... 18
2.15 PHY Receiver Resume ..... 18
2.16 PHY PPDU Reception Stop ..... 19
2.17 PHY PPDU Buffer Release ..... 19
2.18 PHY RX Process ..... 20
2.19 PHY Receiver Packet Decoding Callback Function Registration ..... 20
2.20 PHY Receiver Timing Callback Function Registration ..... 21
2.21 PHY Receiver Parameters Set ..... 22
2.22 PHY Receiver Parameters Get ..... 23
2.23 PHY RX Get Statistics ..... 24
3.0 References ..... 25

# Table of Figures

Figure 1 Overall G3 Software Architecture.....5

# Table of Tables

Table 1 PHY Layer Public APIs.....6

# 1.0 Introduction

This document describes the G3 physical layer software interfaces.. shows the overall G3 SW Stack block diagrams. The PHY Layer includes the following modules:

- PHY Receive Manager (PRXM)
- PHY Transmit Manager (PTXM)
- Rx/Tx Synchronization Module
- MAC/PHY Interface Module
- Common Math Function Utility Module

In HAL Layer AFE driver module is used directly interfacing with the PHY Layer.

All APIs described in this document are C callable.

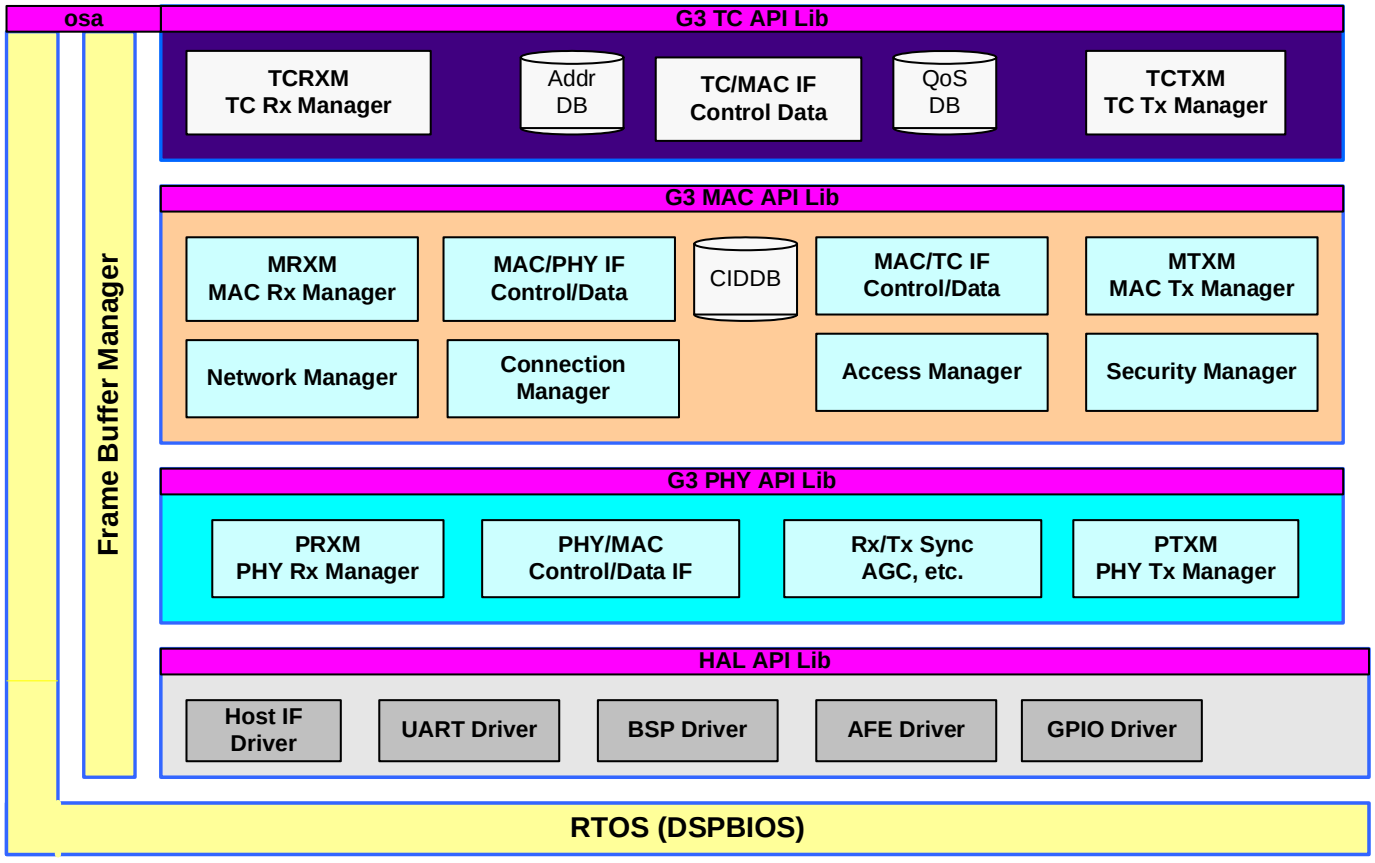


**Figure 1 Overall G3 Software Architecture**

# 2.0 G3 PHY Layer Public APIs

Table 1 lists all the PHY layer library public API definitions.

| API | Sync/Async | Descriptions |
|---|---|---|
| PHY_getLibVersion | Sync | Get PHY library version |
| PHY_init | Sync | Initialize PHY band parameters |
| PHY_txInit | Sync | Initialize PHY Tx chain. |
| PHY_rxInit | Sync | Initialize PHY Rx chain |
| PHY_rxStart | Async | Starts initial preamble detection and synchronization |
| PHY_rxBitStartIndicate | Async | Registers callback for packet decoding |
| PHY_txPpdu | Async | Transmit PHY PPDU to the power line |
| PHY_txAck | Async | Transmit ACK frame to power line |
| PHY_rxPpduStart | Async | Starts PPDU reception process |
| PHY_rxPpduStop | Sync | Stops PPDU reception process |
| PHY_rxPpduRelease | Sync | Release RX PPDU buffer |
| PHY_rxSuspend | Async | Suspend PHY RX operations |
| PHY_rxResume | Sync | Resume PHY suspended RX operations |
| PHY_txSmRun | Sync | Runs PHY TX processing |
| PHY_rxSmRun | Sync | Runs PHY Rx processing |
| PHY_txSet | Sync | Set PHY Tx parameters |
| PHY_rxSet | Sync | Set PHY Rx parameters |
| PHY_txGet | Sync | Get PHY Tx parameters |
| PHY_rxGet | Sync | Get PHY Rx parameters |
| PHY_txGetStatistic | Sync | Read PHY TX statistical parameters |
| PHY_rxGetStatistic | Sync | Read PHY RX statistical parameters |

**Table 1 PHY Layer Public APIs**

NOTE: All asynchronous APIs require callback function as parameter for caller to provide.

The following data structure is defined as common PHY layer public API return code:

```
typedef enum
{
    PHY_STAT_SUCCESS                    = 0,
    PHY_STAT_FAILURE                    = 1,
```

```
    PHY_STAT_PREAMBLE_NOT_DETECTED    = 2,
    PHY_STAT_HEADER_CRC_FAILED        = 3,
    PHY_STAT_PAYLOAD_CRC_FAILED       = 4,
    PHY_STAT_ILLEGAL_PARAMETERS       = 5,
    PHY_STAT_ILLEGAL_OPERATIONS       = 6,
    PHY_STAT_UNKOWN_ID                = 7,
    PHY_STAT_TX_LATE                  = 8,
    PHY_STAT_INVALID_LEN              = 9,
    PHY_STAT_INVALID_SCH              = 10,
    PHY_STAT_INVALID_LEV              = 11,
    PHY_STAT_BUF_OVRUN                = 12,
    PHY_STAT_BUSY                     = 13,
    PHY_STAT_CMD_IN_PLACE             = 14,
    PHY_STAT_NOT_IN_SYNC              = 15
    PHY_STAT_RX_BUF_OVERRUN           = 16

} PHY_status_t;
```

The callback function used for asynchronous APIs for notification is defined as:

```
typedef void (*PHY_cbFunc_t)(PHY_ev_t eventID, PHY_cbData_t *cbData_p);

typedef enum
{
            PHY_EV_RX_START_DONE          = 0,
            PHY_EV_TX_PPDU_DONE           = 1
            PHY_EV_RX_PPDU_DONE           = 2,
            PHY_EV_TX_SUSPEND_DONE        = 3,
            PHY_EV_RX_SUSPEND_DONE        = 4,
            PHY_EV_TX_TESTMODE_DONE       = 5,      // reserved
            PHY_EV_TX_ACK_DONE            = 6,
            PHY_EV_RX_ACK_DONE            = 7,
            PHY_EV_RX_BIT_START           = 8,
            PHY_EV_TX_PPDU_START          = 9,
            PHY_EV_RX_FCH_DONE            = 10,
            PHY_EV_RX_PKT_RCV_DONE        = 11
} PHY_ev_t;


typedef struct
{
 PHY_status_t       status;    // callback status
 union
 {
   PHY_cbTxPpdu_t    txPpdu;    // PHY Tx ppdu done callback
   PHY_cbTxAck_t     txAck;     // PHY Tx ACK frame done callback
   PHY_cbRxSync_t    rxSync;    // PHY Rx channel acquisition callback
   PHY_cbRxPpdu_t    rxPpdu;    // PHY Rx ppdu callback
   PHY_cbRxPpduTiming_t  rxPpduTiming; // PHY Rx ppdu timing
   PHY_cbRxFch_t     rxFch;     // PHY Rx FCH information
   PHY_cbRxAck_t     rxAck ;    // PHY RX ACK frame callback

 }cbParms;

}PHY_cbData_t;

/* Tx PPDU done callback data structure */
typedef struct
{
 UINT32      ppduAddr;        // Tx PPDU address passed by caller

}PHY_cbTxPpdu_t;

/* Rx channel acquired callback data structure */
typedef struct
{
 SINT16      rssi;            // rssi in dBm
 UINT16      nSymbols;        // number of symbols for ppdu

}PHY_cbRxSync_t;
```

```
/* Rx PPDU callback data structure */
typedef struct
{
  UINT32       ppduInfoAddr;       // Rx PPDU info address (see PHY_rxPpdu_t in phy_rx.h)

}PHY_cbRxPpdu_t;


/* Rx FCH callback data structure */
typedef struct
{
   UINT16       num_symbols;  // 695us* num_symbols is the duration of the packet payload
   UINT32       time;         // time stamp of the last symbol of FCH
}PHY_cbRxFch_t;

/* Rx PPDU timing callback data structure */
typedef struct
{
   UINT32       time;         // time stamp of the last symbol of Data packet
}PHY_cbRxPpduTiming_t;


/* Rx ACK frame callback data structure */
typedef struct
{
   UINT16       ack;       // 1 = ACK, 0 = NAK
   UINT16       fcs;       // received FCS (signature for the ACK/NAK frame)

}PHY_cbRxAck_t;


/* Tx ACK frame callback data structure */
typedef struct
{
  UINT32       ackFrmAddr;       // Tx ACK frame address
}PHY_cbTxAck_t;
```

## 2.1 PHY_getLibVersion

This API gets PHY library version string

**Syntax** `const char *PHY_getLibVersion(void)`

| Parameter | Description |
|---|---|
| **Return Value** | Pointer to the PHY library version string.<br>const char PHY_LIB_VERSION[16] |

## 2.2 PHY_init

This API initialize band related parameters

**Syntax** `PHY_status_t char *PHY_init(void)`

| Parameter | Description |
|---|---|
| **Return Value** | PHY_status_t:<br>PHY_STAT_SUCCESS<br>PHY_STAT_FAILURE |

## 2.3 PHY Tx Path Initialization

This API initializes or resets the PHY TX Manager (PHY_TX) module. It's only called once after the unit powered up or system reset.

**Syntax** `PHY_status_t PHY_txInit(void)`

| Parameter | Description |
|---|---|
| **Return Value** | PHY_status_t:<br>PHY_STAT_SUCCESS<br>PHY_STAT_FAILURE |

## 2.4 PHY Transmit Prepare PPDU

This API generates the preamble for a PPDU. It is called by MAC TX Manager (MTXM) every time before a PPDU is transmitted.

**Syntax**
```
PHY_status_t PHY_txPreparePpdu(PHY_tx_ppdu_t *ppdu_p,
                               PHY_cbFunc_t   cb_p)
```

| Parameter | Description |
|---|---|
| Ppdu_p | Pointer to the PPDU primitive data structure configured by caller which contains data buffer for PPDU to be transmitted, modulation, coding scheme and transmit power level for PPDU transmission. Please note only "level" is used during the preparation. Other parameters including payload do not need to be valid when PHY_txPreparePpdu is called.<br>typedef struct<br>{<br>UINT16 *ppdu_p; // pointer to PPDU payload (16-bit word aligned)<br>UINT16 length; // PPDU buffer length in bytes<br>UINT16 level; // TX signal level<br>// Range 0 (minimum) to 0x20 (maximum)<br>UINT16 mcs; // Tx modulation coding scheme<br>// 0: ROBO<br>// 1: DBPSK<br>// 2: DQPSK<br>UINT16 toneMap; // b0 to b8: each bit represents 6 tones,e.g. b0 represents valid<br>// tones 0 to 5<br>UINT16 txGain[2]; // Transmit gain<br>// word 0: b0-3: number of gain step for 20-30 kHz |

| | |
|---|---|
| | //  b4-7  number of gain step for 10-20 kHz<br>//  b8-11  number of gain step for 40-50 kHz<br>//  b12-15  number of gain step for 30-40 kHz<br>// word 1: b0-3:  number of gain step for 60-70 kHz<br>//  b4-7  number of gain step for 50-60 kHz<br>//  b8-11  number of gain step for 80-90 kHz<br>//  b12-15  number of gain step for 70-80 kHz<br><br>UINT16  dt;  // delimiter type<br>// 0 = start of frame with no response expected<br>// 1 = start of frame with response expected<br><br>UINT16  rpt;  // Retransmission<br>// 0 – new packet; 1- retransmit the previous packet<br>} PHY_tx_ppdu_t; |
| cb_p | Callback function when PPDU transmit finishes. It contains the following parameters:<br>eventID =  PHY_EV_TX_PPDU_DONE;<br>cbData.status =  PHY_STAT_SUCCESS  // successful transmission<br>=  PHY_STAT_FAILURE  // Failure transmission<br>cbData.cbParms:<br>typedef struct<br>{<br>UINT32  ppduAddr;  // Tx PPDU address passed by caller<br><br>}PHY_cbTxPpdu_t; |
| **Return Values** | PHY_status_t:<br>PHY_STAT_SUCCESS<br>PHY_STAT_FAILURE |

## 2.5 PHY Transmit PPDU

This API starts a transmission of a PPDU (maximum 239 bytes) to the power line IF. It is called by MAC TX Manager (MTXM) every time when there is a PPDU to be transmitted.

**Syntax**

```
PHY_status_t PHY_txPpdu(PHY_tx_ppdu_t *ppdu_p,
                        PHY_cbFunc_t   cb_p)
```

| Parameter | Description |
|---|---|
| Ppdu_p | Pointer to the PPDU primitive data structure configured by caller which contains data buffer for PPDU to be transmitted, modulation, coding scheme and transmit power level for PPDU transmission.<br>typedef struct<br>{<br>UINT16 *ppdu_p; // pointer to PPDU payload (16-bit word aligned)<br>UINT16 length; // PPDU buffer length in bytes<br>UINT16 level; // TX signal level<br>// Range 0 (minimum) to 0x20 (maximum)<br><br>UINT16 mcs; // Tx modulation coding scheme<br>// 0: ROBO<br>// 1: DBPSK<br>// 2: DQPSK<br><br>UINT16 toneMap; // b0 to b8: each bit represents 6 tones,e.g. b0 represents valid<br>// tones 0 to 5<br>UINT16 txGain[2]; // Transmit gain<br>// word 0: b0-3: number of gain step for 20-30 kHz<br>// b4-7 number of gain step for 10-20 kHz<br>// b8-11 number of gain step for 40-50 kHz<br>// b12-15 number of gain step for 30-40 kHz<br>// word 1: b0-3: number of gain step for 60-70 kHz<br>// b4-7 number of gain step for 50-60 kHz<br>// b8-11 number of gain step for 80-90 kHz<br>// b12-15 number of gain step for 70-80 kHz<br><br>UINT16 dt; // delimiter type<br>// 0 = start of frame with no response expected<br>// 1 = start of frame with response expected<br><br>UINT16 rpt; // Retransmission<br>// 0 – new packet; 1- retransmit the previous packet<br>} PHY_tx_ppdu_t; |
| cb_p | Callback function when PPDU transmit finishes. It contains the following parameters:<br>eventID = PHY_EV_TX_PPDU_DONE;<br>cbData.status = PHY_STAT_SUCCESS // successful transmission<br>= PHY_STAT_FAILURE // Failure transmission<br>cbData.cbParms:<br>typedef struct<br>{ |

| | |
|---|---|
| | UINT32 ppduAddr; // Tx PPDU address passed by caller<br><br>}PHY_cbTxPpdu_t; |
| **Return Values** | PHY_status_t:<br>PHY_STAT_SUCCESS<br>PHY_STAT_FAILURE<br>PHY_STAT_INVALID_LEN<br>PHY_STAT_INVALID_SCH<br>PHY_STAT_INVALID_LEV<br>PHY_STAT_BUSY |

## 2.6 PHY Transmit Prepare ACK Frame

This API generates the preamble for an ACK frame. It is called by MAC TX Manager (MTXM) every time before an ACK is transmitted.

**Syntax**

```
PHY_status_t PHY_txPrepareAck(PHY_tx_ack_t *ack_p,
                              PHY_cbFunc_t   cb_p)
```

| Parameter | Description |
|---|---|
| ack_p | Pointer to the ACK packet [1]. Please note only "level" is used during the preparation. Other parameters do not need to be valid when PHY_txPrepareAck is called.<br>typedef struct<br>{<br>UINT16 ack; // 1 = ACK, 0 = NAK<br>UINT16 fcs; // MAC FCS field received in MAC frame for the ACK/NAK<br>// response<br>UINT16 level; // TX level signal level<br>// range 0 to 0x20 (unit TBD)<br>UINT16 txGain[2]; // Transmit gain<br>// word 0: b0-3: number of gain step for 20-30 kHz<br>// b4-7 number of gain step for 10-20 kHz<br>// b8-11 number of gain step for 40-50 kHz<br>// b12-15 number of gain step for 30-40 kHz<br>// word 1: b0-3: number of gain step for 60-70 kHz<br>// b4-7 number of gain step for 50-60 kHz<br>// b8-11 number of gain step for 80-90 kHz<br>// b12-15 number of gain step for 70-80 kHz<br><br>} PHY_tx_ackFrm_t; |
| cb_p | Callback function when PPDU transmit finishes. It contains the following parameters:<br>eventID = PHY_EV_TX_ACK_DONE;<br>cbData.status = PHY_STAT_SUCCESS // successful transmission<br>= PHY_STAT_FAILURE // Failure transmission |
| **Return Values** | PHY_status_t: |

[1] Note that no MCS or tone map is specified for the ACK frame transmission as it contains only FCH where super robust mode is used and all tones are used.

| | |
|---|---|
| | PHY_STAT_SUCCESS<br>PHY_STAT_FAILURE<br>PHY_STAT_BUSY |

## 2.7 PHY Transmit ACK Frame

This API starts a transmission of an ACK frame to the power line IF. It is called by MAC TX Manager (MTXM) every time when there is an ACK frame to be transmitted.

**Syntax**

```
PHY_status_t PHY_txAck(PHY_tx_ack_t *ack_p,
                       PHY_cbFunc_t   cb_p)
```

| Parameter | Description |
|---|---|
| ack_p | Pointer to the ACK packet [2]<br>typedef struct<br>{<br>UINT16 ack; // 1 = ACK, 0 = NAK<br>UINT16 fcs; // MAC FCS field received in MAC frame for the ACK/NAK // response<br>UINT16 level; // TX level signal level<br>// range 0 to 0x20 (unit TBD)<br>UINT16 txGain[2]; // Transmit gain<br>// word 0: b0-3: number of gain step for 20-30 kHz<br>// b4-7 number of gain step for 10-20 kHz<br>// b8-11 number of gain step for 40-50 kHz<br>// b12-15 number of gain step for 30-40 kHz<br>// word 1: b0-3: number of gain step for 60-70 kHz<br>// b4-7 number of gain step for 50-60 kHz<br>// b8-11 number of gain step for 80-90 kHz<br>// b12-15 number of gain step for 70-80 kHz<br><br>} PHY_tx_ackFrm_t; |
| cb_p | Callback function when PPDU transmit finishes. It contains the following parameters:<br>eventID = PHY_EV_TX_ACK_DONE;<br>cbData.status = PHY_STAT_SUCCESS // successful transmission<br>= PHY_STAT_FAILURE // Failure transmission |
| **Return Values** | PHY_status_t:<br>PHY_STAT_SUCCESS<br>PHY_STAT_FAILURE<br>PHY_STAT_BUSY |

## 2.8 PHY TX Process

[2] Note that no MCS or tone map is specified for the ACK frame transmission as it contains only FCH where super robust mode is used and all tones are used.

This API runs PHY Tx process during active packet transmission. It is to be called at G3 PHY symbol rate (per DMA interrupt, for symbol processing) and once TX bit processing needs to start (for bit processing).

**Syntax** `PHY_status_t PHY_txSmRun(UINT16 mode)`

| Parameter | Description |
|---|---|
| **mode** | TX process mode:<br>0 – TX bit processing;<br>1 – TX symbol processing. |
| **Return Values** | PHY_status_t:<br>PHY_STAT_SUCCESS<br>PHY_STAT_FAILURE |

## 2.9 PHY TX Parameter Set

This API sets PHY TX parameters.

**Syntax** `PHY_status_t PHY_txSet(UINT16 setType, PHY_txSetData_t *setData_p)`

| Parameter | Description |
|---|---|
| setType | Parameter type to be set:<br>0x0000 - PHY_TX_SET_TONE_MASK : set tonemask<br>0x0001 – PHY_TX_SET_BLKILV: set block interleaver<br>0x0002 - PHY_TX_SET_COH : set coherent modulation mode<br>0x0003 – 0xFFFF: reserved |
| setData_p | Pointer to data to be set:<br>typedef union<br>{<br>UINT8 toneMask[12];<br>UINT16 blkIlv;<br>UINT16 cohMod;<br>}PHY_txSetData_t;<br><br>**PHY_TX_SET_TONE_MASK: Set spectral mask (static) where tones start, stop and notch**<br>UINT8 toneMask[0].MSB: 0- Cenelec band 1- FCC band<br>UINT8 toneMask[0] bit6-bit0: number of tones in the band<br>UINT8 toneMask[1]: tone number for the first tone in the band<br>UINT8 toneMask[2] – [11]: tone mask, 1 indicates the tone is on, 0 indicates the tone is off.<br>Detailed example is shown in the table below.<br><br>**PHY_TX_SET_BLKILV: Set to use block interleaver**<br>UINT16 blkIlv: 1-use block interleaver; 0-use whole-packet interleaver<br>**PHY_TX_SET_COH: Set to use coherent modulation mode**<br>UINT16 cohMod: 1-use coherent modulation; 0-use differential modulation |

| Return Values | PHY_status_t:<br>PHY_STAT_SUCCESS<br>PHY_STAT_FAILURE |
|---|---|

Tone mask definition:

| Band | Tone mask |
|---|---|
| **Cenelec A 36** | 24.17.ff.ff.ff.ff.0f.00.00.00.00.00 |
| **Cenelec A 25** | 24.17.ff.ff.00.f8.0f.00.00.00.00.00 |
| **Cenelec B** | 10.3f.ff.ff.00.00.00.00.00.00.00.00 |
| **Cenelec BC** | 1a.3f.ff.ff.ff.03.00.00.00.00.00.00 |
| **Cenelec BCD** | 20.3f.ff.ff.ff.ff.00.00.00.00.00.00 |
| **FCC low band** | a4.1f.ff.ff.ff.ff.0f.00.00.00.00.00 |
| **FCC high band** | a4.43.ff.ff.ff.ff.0f.00.00.00.00.00 |
| **FCC full band** | c8.1f.ff.ff.ff.ff.ff.ff.ff.ff.ff.00 |

## 2.10 PHY TX Parameter Get

This API gets PHY TX parameters.

**Syntax** `PHY_status_t` **`PHY_txGet`**`(UINT16 getType, PHY_txGetData_t *getData_p)`

| Parameter | Description |
|---|---|
| getType | Parameter type to get:<br>0x0000 – PHY_TX_GET_TONE_MASK: Get tone mask<br>0x0001 – PHY_TX_GET_BLKILV: Get block interleaver flag<br>0x0002 – PHY_TX_GET_ZCT: Get last zero crossing time<br>0x0003 – PHY_TX_GET_TIME: Get PHY time (in 10us)<br>0x0004 - PHY_TX_GET_NUMINFOTONES: get number of information tones in payload<br>0x0005 - PHY_TX_GET_NUMFCHSYMBS: get number of FCH symbols<br>0x0006 - PHY_TX_GET_COH: get coherent modulation mode<br>0x0007 – 0xFFFF: reserved |
| getData_p | Pointer to the storage where the results of the corresponding getType stores:<br>typedef union<br>{<br>PHY_zcTime_t zcTime; // Zero crossing time<br>UINT16 toneMask[6]; // Start, stop, notches<br>UINT16 blkIlv; // Block interleaver flag<br>UINT32 currTime; // Current PHY time (in 10us)<br>UINT32 numInfoTones; // number of information tones in payload<br>UINT16 numFCHSymbs; // number of FCH symbols<br>UINT16 cohMod; // coherent modulation<br>}PHY_txGetData_t;<br><br>**PHY_TX_GET_TONE_MASK: Get spectral mask (static) where tones start, stop and notch , see PHY_TX_SET**<br>UINT16 toneMask[6]: see tone mask definition in 2.7 |

| | |
|---|---|
| | **PHY_TX_GET_BLKILV: Get block interleaver flag**<br>UINT16 blkIlv: 1-use block interleaver; 0-use whole packet interleaver<br><br>**PHY_TX_GET_ZCT: Get last zero-crossing time (in 10us)**<br>typedef struct<br>{<br>UINT32 lastZcaTime; // last zero crossing time in 10us (20-bits)<br>UINT32 lastZcbTime; // last zero crossing time in 10us (20-bits)<br>}PHY_zcTime_t;<br>lastZcaTime = time of last zero crossing event for phase A (in 10us)<br>lastZcbTime = time of last zero crossing event for phase B (in 10us)<br><br>**PHY_TX_GET_TIME: Get current PHY time**<br>: UINT32 currTime: current PHY time (in 10us)<br><br>**PHY_TX_GET_NUMINFOTONES: Get number of information tones based on tone mask and tone map**<br>UINT32 numInfoTones: this parameter is used as both input and output parameter.<br>use as input: tone map definition<br>use as output: number of information tones in a payload<br><br>**PHY_TX_GET_NUMFCHSYMBS: get number of FCH symbols**<br>UINT16 numFCHSymbs: number of FCH symbols in a packet<br><br>**PHY_TX_GET_COH: Get coherent modulation mode**<br>UINT16 cohMod: 1-use coherent modulation; 0-use differential modulation |
| **Return Values** | PHY_status_t:<br>PHY_STAT_SUCCESS<br>PHY_STAT_FAILURE |

## 2.11 PHY TX Get Statistics

This API reads PHY TX path statistics.

**Syntax** `PHY_status_t PHY_txGetStat(PHY_tx_stat_t *getData_p)`

| Parameter | Description |
|---|---|
| getData_p | Pointer to the buffer which can be used to store the returned PHY Tx statistics:<br>Typedef struct {<br>UINT16 phyStatsTxDropCount; // # of times Tx PHY drops data<br>UINT32 phyTxProcDelay; // time in 10's us between TxPpdu() to PPDU goes to line IF<br>} PHY_tx_stat_t; |
| **Return Values** | PHY_status_t:<br>PHY_STAT_SUCCESS<br>PHY_STAT_FAILURE |

## 2.12 PHY Layer Rx Path Initialization

This function initializes or resets the PHY Rx Manager (PHY_RX) module. It initializes the PHY receiver global data structure, internal states and reset buffers. It is called once at power up or reset.

**Syntax** `PHY_status_t PHY_rxInit(void)`

| Parameter | Description |
|---|---|
| **Return Values** | PHY_status_t:<br>PHY_STAT_SUCCESS<br>PHY_STAT_FAILURE |

## 2.13 PHY Receiver Start Channel Acquisition

This function requests PHY to start the initial synchronization or resynchronization to the power line channel. This includes preamble detection and valid header parsing. This should be called at least once after power up.

**Syntax**

```
PHY_status_t PHY_rxStart(UINT16 timeOut,
                         PHY_cbFunct_t cb_p)
```

| Parameter | Description |
|---|---|
| timeOut | initial synchronization time out values (in number of symbols). 0xFFFF means never time-out. Caller has to use PHY_rxSuspend() to stop the receiver synchronization process before time-out happens. |
| cb_p | Callback function when a complete PPDU data unit has been received from power line. It contains the following parameters:<br>eventID: PHY_EV_RX_PPDU_ DONE // when packet is received<br>PHY_EV_RX_ACK_DONE // when ACK frame is received<br><br>cbData.status: // PHY PPDU receive status code which contains<br>PHY_STAT_SUCCESS<br>PHY_STAT_RX_BUF_OVERRUN<br><br>cbData.cbParms. // for **eventID = PHY_EV_RX_PPDU_DONE:**<br>PHY_cbRxPpdu_t rxPpdu;<br><br>typedef struct<br>{<br>UINT32 ppduInfoAddr; // Rx PPDU info address (see PHY_rxPpdu_t in phy_rx.h)<br>} PHY_cbRxPpdu_t<br><br>ppduInfoAddr: received PPDU data structure start address (buffer owned by PHY). The PPDU format is as following:<br>typedef struct {<br>UINT16 id; // ppdu ID<br>UINT16 length; // number of PSDU bytes<br>UINT16 lqi; // link quality indicator (SNR)<br>UINT16 level; // received level:<br>// 0: <= 70dBuV |

| | |
|---|---|
| | // 1: <= 72dBuV<br>// 2: <= 74dBuV<br>// …..<br>// 15: > 98dBuV<br>UINT16 mcs; // modulation coding scheme:<br>// 0: ROBO<br>// 1: DBPSK<br>// 2: DQPSK<br><br>UINT16 *ppdu_data_p; // pointer to PSDU data<br>SINT16 *sb_snr_p; // pointer to subband SNR for the current packet<br>} PHY_rxPpdu_t<br><br>cbData.cbParms: // for **eventID = PHY_EV_RX_ACK_DONE:**<br>PHY_cbRxAck_t *field1;<br><br>typedef struct<br>{<br>UINT16 ack; // 1 = ACK, 0 = NAK<br>UINT16 fcs; // received FCS (signature for the ACK/NAK frame)<br>} PHY_cbRxAck_t<br><br>A NULL pointer means current PPDU reception contains errors (reference statusCode).<br><br>The memory storage used for received PPDU and its relative info is own by PHY. It is caller's responsibility to release them through PHY_rxPpduRelease() before PHY can use them again for next packet receive. |
| **Return Values** | PHY_status_t:<br>PHY_STAT_SUCCESS<br>PHY_STAT_FAILURE<br>PHY_STAT_CMD_IN_PLACE // PHY Rx is already in sync |

## 2.14 PHY Receiver Suspend

This function requests to suspend all PHY receiver's present activities including all reception functions.

**Syntax** `PHY_status_t PHY_rxSuspend(PHY_cbFunct_t cb_p)`

| Parameter | Description |
|---|---|
| cb_p | Callback function when PHY_rxSuspend() is completed. It has the following parameters:<br>eventID: PHY_EV_RX_SUSPEND_DONE; |
| **Return Values** | PHY_status_t<br>PHY_STAT_SUCCESS<br>PHY_STAT_FAILURE<br>PHY_STAT_CMD_IN_PLACE |

## 2.15 PHY Receiver Resume

This function requests to resume PHY receiver's suspended activities. PHY shall start its normal reception. More details TBD.

**Syntax** `PHY_status_t PHY_rxResume(void)`

| Parameter | Description |
|---|---|
| **Return Value** | PHY_status_t:<br>PHY_STAT_SUCCESS,<br>PHY_STAT_FAILURE,<br>PHY_STAT_CMD_IN_PLACE |

## 2.16PHY PPDU Reception Stop

This function can only be called after a PHY PPDU reception process has been started through PHY_rxPpduStart().By calling this function, PHY will stop the delivering of the received streams to the caller. This is a synchronous call.

**Syntax** `PHY_status_t PHY_rxPpduStop(void)`

| Parameter | Description |
|---|---|
| Return Values | PHY_status_t :<br>PHY_STAT_SUCCESS<br>PHY_STAT_FAILURE<br>PHY_STAT_CMD_IN_PLACE // no active PPDU reception in progress |

## 2.17 PHY PPDU Buffer Release

This function is called every time after PHY delivers a received PPDU from the power line IF to the caller (through the callback function registered) to indicate to the PHY that the buffer used to store the PPDU is available to PHY again. If the buffer is not released in time, the PHY will issue a callback function with NULL PPDU buffer and signal PHY Rx buffer overrun error. This is a synchronous API.

**Syntax** `PHY_status_t PHY_rxPpduRelease(PHY_rxPpdu_t *ppdu_p)`

| Parameter | Description |
|---|---|
| ppdu_p | PPDU buffer pointer to be released. PPDU buffer format:<br>typedef struct {<br>UINT16 id; // ppdu ID<br>UINT16 length; // number of PSDU bytes |

| | |
|---|---|
| | UINT16 level; // received level:<br>// 0: <= 70dBuV<br>// 1: <= 72dBuV<br>// 2: <= 74dBuV<br>// …..<br>// 15: > 98dBuV<br>UINT16 mcs; // modulation coding scheme:<br>// 0: ROBO<br>// 1: DBPSK<br>// 2: DQPSK<br><br>UINT16 *ppdu_data_p; // pointer to PSDU data<br>UINT32 time; // time when preamble is received<br>} PHY_rxPpdu_t |
| Return Values | PHY_status_t :<br>PHY_STAT_SUCCESS<br>PHY_STAT_FAILURE<br>PHY_STAT_CMD_IN_PLACE // buffer already released |

## 2.18PHY RX Process

This API runs PHY Rx process during active packet transmission. It is to be called at G3 PHY symbol rate (per DMA interrupt for symbol processing) and when bit processing starts (for bit processing).

**Syntax** `PHY_status_t PHY_rxSmRun(PHY_rxProc_t procType)`

| Parameter | Description |
|---|---|
| **procType** | Process type.<br><br>typedef enum<br>{<br>PHY_RX_PROC_SYMB = 0, // Symbol Processing<br>PHY_RX_PROC_BIT = 1 // Bit Processing<br><br>}PHY_rxProc_t; |
| **Return Values** | PHY_status_t:<br>PHY_STAT_SUCCESS<br>PHY_STAT_FAILURE |

## 2.19 PHY Receiver Packet Decoding Callback Function Registration

This function registers the receiver for packet decoding callback function. This should only be registered once before channel acquisition start is requested.

Note that this callback needs to be processed at a lower priority than the “PHY Rx Process” described in Section 2.15.

**Syntax** `PHY_status_t PHY_rxBitStartIndicate(PHY_cbFunct_t cb_p)`

| Parameter | Description |
|---|---|
| cb_p | Callback function when bit processing of a PPDU data unit can start. It contains the following parameters:<br>eventID: PHY_EV_RX_BIT_START // when bit processing of packet start<br><br>cbData.status: // PHY receive bit processing status code which contains<br>PHY_STAT_SUCCESS |
| Return Values | PHY_status_t :<br>PHY_STAT_SUCCESS |

## 2.20 PHY Receiver Timing Callback Function Registration

This function registers the receiver for PHY timing callback function. This should only be registered once before channel acquisition start is requested.

**Syntax** `PHY_status_t PHY_rxPhyTimingIndicate (PHY_cbFunct_t cb_p)`

| Parameter | Description |
|---|---|
| cb_p | Callback function when bit processing of a PPDU data unit can start. It contains the following parameters:<br>eventID: PHY_EV_RX_FCH_DONE // when bit processing of FCH is done<br>PHY_EV_RX_PKT_RCV_DONE // when last sample of the RX packet is received<br><br>cbData.status: // PHY receive bit processing status code which contains<br>PHY_STAT_HEADER_CRC_FAILED // FCH is decoded but header CRC fails<br>PHY_STAT_SUCCESS<br><br>cbData.cbParms. // for **eventID = PHY_EV_RX_FCH_DONE:**<br>PHY_cbRxFch_t rxFCH;<br><br>typedef struct<br>{<br>UINT16 num_symbols; // 695us* num_symbols is the duration of the packet payload<br>UINT32 time; // time stamp of the last symbol of FCH<br>}PHY_cbRxFch_t;<br>cbData.cbParms. // for eventID = **PHY_EV_RX_PKT_RCV_DONE**:<br>PHY_cbRxPpduTiming_t rxPpduTiming;<br><br>typedef struct<br>{<br>UINT32 time; // time stamp of the last symbol of Data packet<br>}PHY_cbRxPpduTiming_t; |

| | |
|---|---|
| | |
| Return Values | PHY_status_t :<br>PHY_STAT_SUCCESS |

## 2.21 PHY Receiver Parameters Set

This function sets the values for PHY parameters.

**Syntax**

```
PHY_status_t PHY_rxSet(UINT16 setType,
                       PHY_rxSetData_t *setData_p)
```

| Parameter | Description |
|---|---|
| setType | Parameter type to be set:<br>0x0000: PHY_RX_SET_AGC – set AGC parameters<br>0x0001: PHY_RX_SET_PPDU_INFO – set PPDU parameters<br>0x0002: PHY_RX_SET_BLKILV: set block interleaver<br>0x0003: PHY_RX_SET_TONEMASK : set tonemask<br>0x0004: PHY_RX_SET_COH: set coherent modulation mode<br>0x0004-0xFFFF: reserved |
| setData_p | Pointer to data to be set:<br>typedef union<br>{<br>UINT16 toneMask[6];<br>PHY_rxAagcSet_t aagc;<br>PHY_rxPpduSet_t ppdu;<br>UINT16 blkIlvMode;<br>UINT16 cohMod;<br>}PHY_rxSetData_t;<br><br>**PHY_SET_AGC: set AGC parameters**<br>typedef struct<br>{<br>UINT16 mode; // AGC mode = 0 (auto), 1(manual)<br>UINT16 step; // AGC gain step = 0 to N-1 where N is maximum number of AGC gain steps (in increment of AGC gain_step_dB)<br><br>}PHY_rxAagcSet_t;<br>**PHY_RX_SET_TONEMASK: Set spectral mask (static) where tones start, stop and notch**<br>See tone mask definition in 2.7<br>**PHY_RX_SET_BLKILV: Set to use block interleaver**<br>UINT16 blkIlv: 1-use block interleaver; 0-use whole-packet interleaver<br><br>**PHY_RX_SET_COH: Set to use coherent modulation mode**<br>UINT16 cohMod: 1-use coherent modulation; 0-use differential modulation |
| **Return Values** | PHY_status_t:<br>PHY_STAT_SUCCES<br>PHY_STAT_FAILURE |

## 2.22 PHY Receiver Parameters Get

This function gets the PHY receiver parameter values.

**Syntax** `PHY_status_t PHY_rxGet(UINT16 getType, UINT16 *getData_p)`

| Parameter | Description |
|---|---|
| getType | Parameter type to be get:<br>0x0000: PHY_GET_AGC<br>0x0001: PHY_GET_CD<br>0x0002: PHY_GET_NL<br>0x0003: PHY_GET_SNR<br>0x0004: PHY_GET_RQ<br>0x0005: PHY_GET_SBSNR<br>0x0006: PHY_RX_GET_TONE_MASK<br>0x0007: PHY_RX_GET_COH |
| getData_p | stores the results corresponding getType:<br>typedef union<br>{<br>PHY_rxAagcGet_t aagc;<br>PHY_rxCdGet_t cd;<br>UINT16 nl; // noise level 0 to 15 (50 to 92 dBuV in 3 dB step)<br>UINT16 snr; // snr 0 tp 7 (0 to 18 dB in 3 dB step)<br>UINT16 rq; // receive quality (0 to 8 from bad to good)<br>UINT16 sb_snr[10]; // SNR for subbands<br>UINT16 cohMod;<br>}PHY_rxGetData_t;<br>**PHY_GETS_AGC:** get AGC parameters<br>typedef struct<br>{<br>UINT16 mode; // AGC mode = 0 (auto), 1(manual)<br>UINT16 step; // 0 to N-1 where N is maximum number of AGC gain<br>}PHY_rxAagcGet_t;<br><br>**PHY_GET_CD:** get carrier detection status<br>UINT16 busy; // 1 = busy<br>// 0 = Idle<br><br>**PHY_GET_NL:** get noise floor level<br>UINT16 nl: // noise floor level, it contains one of following numbers:<br>// 0: <= 50dBuV<br>// 1: <= 53dBuV<br>// 2: <= 56dBuV<br>// …..<br>// 15: > 92dBuV<br>**PHY_GET_SNR:** ratio of measured received signal level to noise level of last received PPDU:<br>UINT16 snr: // SNR of last received PPDU, it contains one of following numbers:<br>// 0: <= 0 dB<br>// 1: <= 3 dB<br>// 2: <= 6 dB<br>// ….. |

| | |
|---|---|
| | // 7: > 18 dB<br>**PHY_GET_RQ:** last received frame quality<br>UINT16 rq: // quality of the received signal, measured by the EVM parameter. It contains<br>// one of the following numbers:<br>// 0: bad<br>// …..<br>// 8: good<br>**PHY_GET_SBSNR:** ratio of measured received signal level to noise level of subbands of last received PPDU:<br>UINT16 sb_snr[10]: // SNR of subbands last received PPDU, it contains one of following numbers:<br>// 0: <= 0 dB<br>// 1: <= 3 dB<br>// 2: <= 6 dB<br>// …..<br>// 7: > 18 dB<br><br>// array index<br>// [0]: SNR for tones no. 11 – 16 for 17kHz-27kHz spectrum<br>// [1]: SNR for tones no. 17 – 22 for 27kHz-36kHz spectrum<br>// [2]: SNR for tones no. 23 – 28 for 36kHz-45kHz spectrum<br>// [3]: SNR for tones no. 29 – 24 for 45kHz-55kHz spectrum<br>// [4]: SNR for tones no. 35 – 40 for 55kHz-64kHz spectrum<br>// [5]: SNR for tones no. 41– 46 for 64kHz-73kHz spectrum<br>// [6]: SNR for tones no. 47 – 52 for 73kHz-82kHz spectrum<br>// [7]: SNR for tones no. 53 – 58 for 82kHz-91kHz spectrum<br>**PHY_RX_GET_TONE_MASK: Get spectral mask (static) where tones start, stop and notch , see PHY_RX_SET**<br>UINT16 toneMask[6]: see tone mask definition in 2.7<br><br>**PHY_RX_GET_COH: Get coherent modulation mode**<br>UINT16 cohMod: 1-use coherent modulation; 0-use differential modulation |
| Return Values | PHY_status_t:<br>PHY_STAT_SUCCESS<br>PHY_STAT_FAILURE |

## 2.23 PHY RX Get Statistics

This API gets PHY RX path statistics.

**Syntax** `PHY_status_t PHY_rxGetStat(PHY_rx_stat_t *getData_p)`

| Parameter | Description |
|---|---|
| getData_p | Pointer to the buffer which can be used to store the returned PHY Rx statistics:<br>typedef struct {<br>UINT16 phyCrcFailCount; // # of bursts received with correct CRC but invalid protocol header<br>UINT16 phyFalseDetCount; // # of preamble false detection<br>UINT16 phyRxDropCount; // # of received PPDU drop owing buffer overrun<br>UINT16 phyRxQueueLen; // # of concurrent MPDUs Rx buffer can hold<br>UINT32 phyRxProcDelay; // time in us from PPDU received from AFE to MPDU available to MAC |

| | |
|---|---|
| | SINT16 phyAgcMinGain; // minimum gain for AGC <0dB<br>UINT16 phyAgcStepValue; // distance between steps in dB < 6dB<br>UINT16 phyAgcStepNum; // number of steps so that phyAgcMinGain+((phAgcStepNum – 1)*phyAgcStepValue) >= 21dB<br>UINT32 phyRxPpduCount; // number of packets received since switched on<br>} PHY_rx_stat_t; |
| **Return Values** | PHY_status_t:<br>PHY_STAT_SUCCESS<br>PHY_STAT_FAILURE |

# 3.0 References


[1] “FSM SDS”
[2] “PLC G3 Physical Layer Specification”