IMST GmbH

Carl-Friedrich-Gauß-Str. 2-4, D-47475 Kamp-Lintfort

Wireless M-Bus Range Extender

Host Controller Interface Protocol

Version 1.2

Document State

wip

Date

13.09.2021

Document ID

4000/40140/0158

History

Version	Date	Comment
1.0	28.09.2020	Initial Version
1.1	23.11.2020	Updates with respect to firmware version 1.0
1.2	13.09.2021	Updates with respect to firmware version 1.1 - Status Message extended by battery voltage (Get Application Status) - Range Extender Configuration Options : new option for WM-Bus message output / upload with RSSI value (Configurable Range Extender Options) - New WM-Bus Packet Format including RSSI value (WM-Bus Packet Notification) - Update to LoRaWAN Stack v1.0.4 - New LoRaWAN Stack Configuration options (Configurable LoRaWAN Stack Settings) SLIP Encoder example code with bufferless design added

Aim of this document

This document includes a description of the Host Controller Interface Protocol which is supported by the WM-Bus Range Extender.

Chapter 1 outlines the general WiMOD HCI Protocol terms and format, which is also used in other products of IMST GmbH.

In Chapter 2 and Chapter 3 the format of the application specific messages is given.

The Appendix includes some example implementation as C/C++ code .

Notation Info

Suffix "b" = binary data

Suffix "h" = hexadecimal data

Without suffix = decimal data

Multi byte / octet fields are considered to be treated as unsigned integers with **L**east **S**ignificant **B**yte first unless explicitly noted

Content

Host Controller Interface Overview

The information exchanged between a Host Controller and the target device is based on serial messages.

There are three different HCI messages used in general:

- **Request Messages**
  These are messages sent from the Host Controller to the connected target device (WM-Bus Range Extender).
- **Response Messages**
  These are the correponding response messages which should be expected by the Host Controller in very short time (less than a second) as a result for a request message.
- **Event Messages**
  These are messages transmitted by the target device as a result of an interrupt.

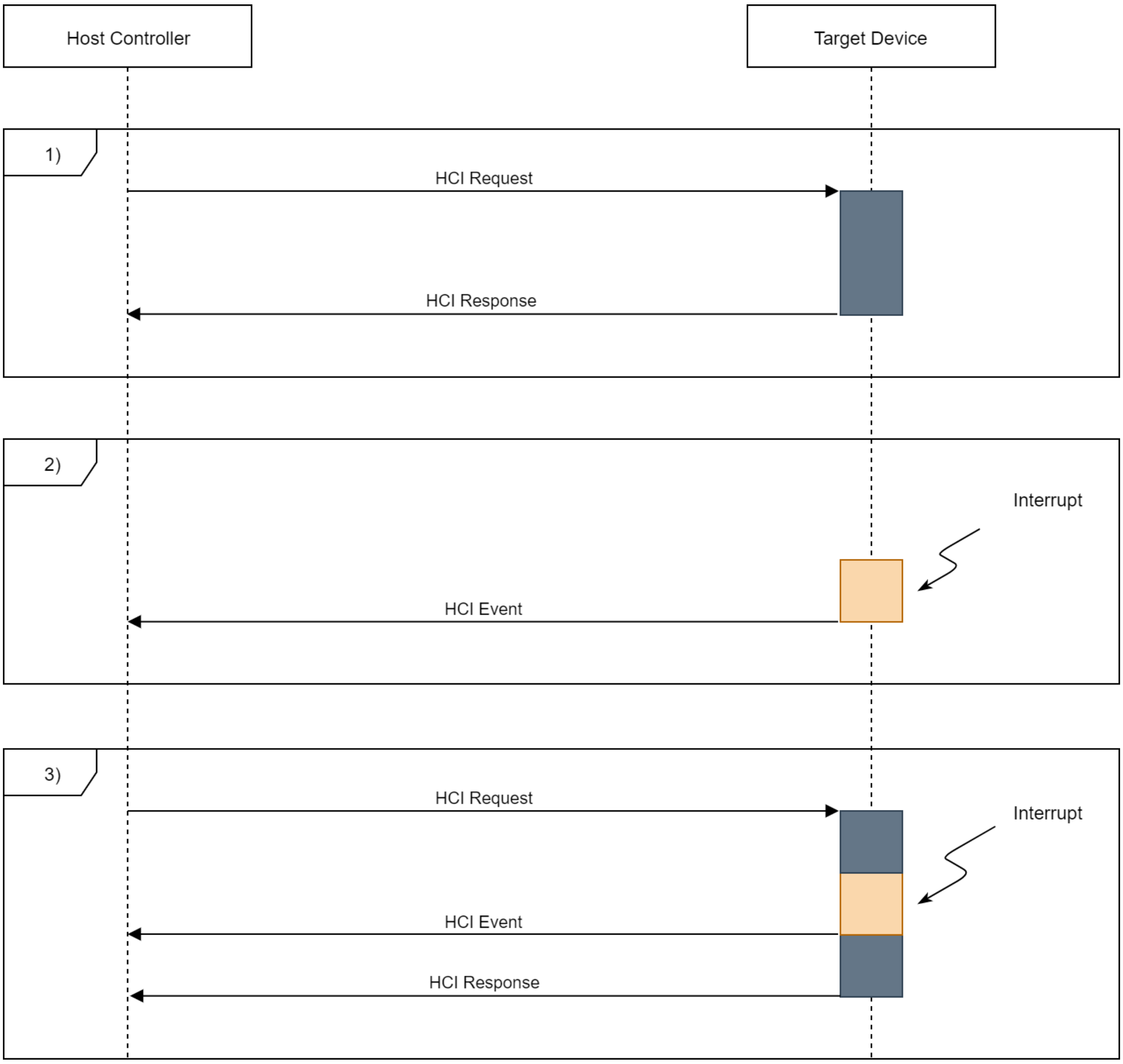

Figure : HCI Message Flow

Note
A Host Controller should be ready to receive event messages at any time.

General Message Format

The next figure outlines the generic message format:

HCI Message			
SAP ID	Msg ID	Message Payload	FCS Field
8 Bit	8 Bit	n * 8 Bit	2 * Bit

Figure : HCI Message Format

A message includes the following fields:

- **Service Access Point Identifier (SAP ID)**
  Identifies a logical message endpoint.

- **Message Identifier (Msg ID)**
  Defines the type of a message.

- **Message Payload**
  The Message Payload field contains optional data. The length of this field is variable (0...max. 500 Octets).

- **FCS Field**
  The **F**rame **C**heck **S**equence field contains a 16-BIT CCITT CRC for bit error detection.

Info
The CRC is helpful in noisy environments and in case of battery powered devices when supply voltages are getting low and single octets may get lost.

Physical Interface

The WM-Bus Range Extender uses a standard UART interface for communication purposes with the following settings:

Baudrate	Start Bits	Data Bits	Parity	Stop Bits	Short
115200 bps	1	8	None	1	8N1

Table : UART Parameters

Framing Protocol

For proper message exchange the widely used SLIP Framing Protocol (https://en.wikipedia.org/wiki/Serial_Line_Internet_Protocol, RFC1055) is implemented. This protocol ensures a secure synchronization between a message sender and a message receiver.

The following figure shows the relationship between a single HCI message and the resulting SLIP message which may include some additional stuffing octets (SLIP ESC) to mark the special reserved SLIP END octets which might occur within a HCI message.

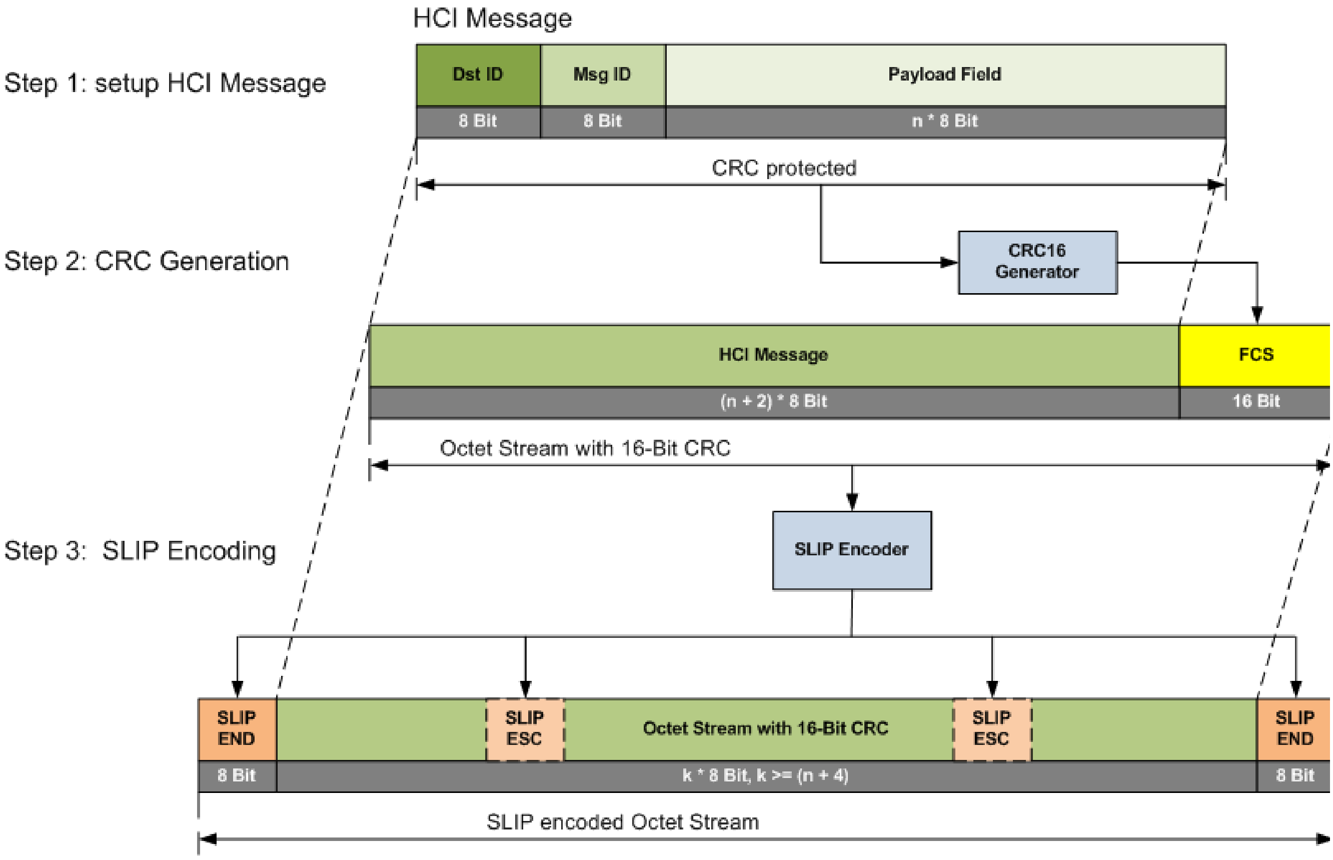

Figure : SLIP Message for communication over UART

Note
The variable length of a message is not explicitly transmitted. Therefore it must be returned as a result form the SLIP decoder.

Back to Top

Application Messages

The supported application messages are grouped to so called **S**ervice **A**ccess **P**oints (SAP).

	HCI Message		
	SAP ID	MSG ID	Payload

Name	SAP ID	Description
Device Management Services	01_h	Provides general services for hardware and firmware identification
WM-Bus Range Extender	07_h	Provides specific services of this application

Table : Service Access Points

Back to Top

Device Management Services

This Service Access Point includes messages for identification and configuration purposes:

Name	Description
Ping	For simple connection test purposes
Get Device Information	Provides hardware related information for identification purposes
Get Firmware Information	Provides firmware specific information for identification purposes
Date and Time Services	Setter and getter for Date & Time
Restart Device	Initiate a device restart

Table : Device Management Services

Back to Application Messages

Ping

This message can be used to test the serial connection between the host controller and the target device. The host should expect a response within a very short time interval.

HCI Message							
Request	**Msg ID**	**Payload**					
	8 Bit	none					
	01_h						
Response	**Msg ID**	**Status**					
	8 Bit	8 Bit					
	02_h	00_h = OK					

Note: the complete SLIP encoded Ping Request and Ping Response messages look like this:

Request : C0 01 01 16 07 C0

Response: C0 01 02 00 A0 AF C0

Back to Device Management Services

Get Device Information

This message can be used to retrieve some information about the hardware of the connected device.

HCI Message						
Request	**Msg ID**	**Payload**				
	8 Bit	none				
	03_h					
Response	**Msg ID**	**Status**	**Module Type**	**Module ID**	**Product Type (optional[1])**	**Product ID (optional[1])**
	8 Bit	8 Bit	8 Bit	32 Bit, LSB First	32 Bit, LSB First	32 Bit, LSB First
	04_h	00_h = ok	$A3_h$ = iM881A-XL	unique ID of embedded radio module	unique product type identifier	unique product identfier (serial number)

[1] provided in firmware version 1.0 and later

Back to Device Management Services

Get Firmware Information

This message can be used to retrieve some information about the firmware of the connected device.

HCI Message						
Request	**Msg ID**	**Payload**				
	8 Bit	none				
	05$_h$					
Response	**Msg ID**	**Status**	**Firmware Version**	**Build Count**	**Build Date**	**Firmware Name**
	8 Bit	8 Bit	2 x 8 Bit, Minor version first	16 Bit, LSB first	10 Octets, ASCII String without terminating "0"	n remaining Octets of message ASCII String, without terminating "0"
	06$_h$	00$_h$ = ok	e.g. (09 00)	e.g. (37 00)$_h$	e.g. (30 39 2E 30 34 2E 32 30 32 30)$_h$	e.g. (57 4D 42 75 73 ... 65 72)$_h$
			=> Version 0.9	=> BC 55	"09.04.2020"	"WMBus_Range_Extender"

Back to Device Management Services

Date and Time Services

The following messages can be used to read and write the current date and time of the connected target device.

Get Date and Time

This message can be used to retrieve the current RTC date and time.

HCI Message			
Request	**Msg ID**	**Payload**	
	8 Bit	none	
	0F$_h$		
Response	**Msg ID**	**Status**	**Date and Time (UTC)**
	8 Bit	8 Bit	32 Bit, LSB first
	10$_h$	00$_h$ = ok	e.g. (19 9E 64 5F)$_h$
			5F649E19$_h$ = 1.600.429.593 seconds since 01.01.1970, 00:00:00 "2020-09-18 11:46:33"

Set Date and Time

This message can be used to configure the embedded RTC.

HCI Message			
Request	**Msg ID**	**Date and Time (UTC)**	
	8 Bit	32 Bit, LSB first	
	0D$_h$	e.g. (19 9E 64 5F)$_h$	
		5F649E19$_h$ = 1.600.429.593 seconds since 01.01.1970, 00:00:00 "2020-09-18 11:46:33"	
Response	**Msg ID**	**Status**	**Payload**
	8 Bit	8 Bit	none
	0E$_h$	00$_h$ = ok	

Back to Device Management Services

Restart Device

This message can be used to initiate a device restart. The device will return a response message immediately and performs a software reset after approx. 200 ms.

HCI Message		
Request	**Msg ID**	**Payload**
	8 Bit	none
	07_h	
Response	**Msg ID**	**Status**
	8 Bit	8 Bit
	08_h	00_h = ok

Back to Device Management Services

WM-Bus Range Extender Services

This Service Access Point includes all application related messages.

Name	Description
Configurable Calendar Events	Setter and getter messages for configuration of Calendar Events
Configurable WM-Bus Device Filter Items	Setter and getter messages for configuration of WM-Bus Device Filter Items
Configurable Range Extender Options	Setter and getter messages for further configurable options like WM-Bus Packet Duplicate Filter
Configurable LoRaWAN Device EUI	Setter and getter messages for configuration of LoRaWAN Device EUI
Configurable LoRaWAN OTAA Settings	Setter and getter messages for configuration of LoRaWAN OTAA parameters
Configurable LoRaWAN ABP Settings	Setter and getter messages for configuration of LoRaWAN ABP parameters
Configurable LoRaWAN Activation Type	Setter and getter messages fo configuration of either OTAA or ABP activation type
Get Application Status	Getter message for Application Status
Application Events	Service to trigger application events like "Enable WM-Bus Recording"
Application Notifications	Notification messages to connected Host Controller
WM-Bus Packet Notification	Notification including received WM-Bus packet
Reset WM-Bus Packet Counter	Service to reset the internal WM-Bus Packet Counter
WM-Bus Range Extender Status Codes	Table of return codes for this Service Access Point

Table : WM-Bus Range Extender Services

Back to Application Messages

Configurable Calendar Events

The following messages are used to read and write the Calendar configuration.

Set Calendar Event List

This message is used to write a complete new Calendar Event configuration.

Note: Any previously configured event will be deleted.

An empty list will also delete all existing events.

HCI Message

Request	**Msg ID**	**Calendar Event Item #1**	**...**	**Calendar Event Item #N**			
	8 Bit	64 Bit	...	64 Bit			
	01_h	see Calendar Event Item, 0 <= N <= 32					
Response	**Msg ID**	**Status**					
	8 Bit	8 Bit					
	02_h	see Status Codes					

Get Calendar Event List

This message is used to read out the current Calendar configuration.

HCI Message

Request	**Msg ID**	**Payload**					
	8 Bit	none					
	03_h						
Response	**Msg ID**	**Status**	**Calender Event Item #1**	**...**	**Calendar Event Item #N**		
	8 Bit	8 Bit	64 Bit	...	64 Bit		
	04_h	see Status Codes	see Calendar Event Item, 0 <= N <= 32				

Calendar Event Item

The next figure outlines the detailed format of a single Calendar Event Item

Calendar Event Item

Event ID	**Filter Group ID**	**Repetition Type**	**Repetition Step Size**	**Date & Time (UTC)**			
8 Bit	8 Bit	8 Bit	8 Bit	32 Bit, LSB first			
see Application Events				see Set Date and Time			

- **Event ID**
  The event type defines the kind of action to be performed. A list of possible Event Types is given here: Application Events

- **Filter Group ID**
  This element is only used in combination with Wireless M-Bus reception / recording types. It defines the group of WM-Bus Filter Items which should be applied during a Wireless M-Bus reception / recording phase.
  Note: The value 255 (FF_h) is reserved and means that all configured Wireless M-Bus Filters should be applied independent of their

own configured Filter Group ID

- **Repetition Type**
  The repetition type defines the periodicity of an event:
  0 = No repetition, single event, can be used for test purpose
  1 = Every Minute
  2 = Hourly
  3 = Daily
  4 = Weekly
  5 = Monthly

- **Repetition Step Size**
  The repetition step size is a second parameter which defines the periodicity of an event:
  Example 1: Repetition Type = 2 (Hourly), Repetion Step Size = 2 => Repetiton Interval = every 2 + 1 = 3 hours
  Example 2: Repetition Type = 3 (Daily), Repetion Step Size = 0 => Repetiton Interval = every 0 + 1 = 1 days

- **Date & Time**
  The date / time element defines when the event should be scheduled for the first time.

Back to WM-Bus Range Extender Servcies

Configurable WM-Bus Device Filter Items

The following messages are used to read and write the WM-Bus Device Filter configuration.

Set WM-Bus Device Filter Item List

This message is used to write a complete new list of WM-Bus Device Filter Items.

Note: Any previously configured filter will be deleted.

An empty list will also delete all existing events.

HCI Message				
Request	**Msg ID**	**WM-Bus Device Filter Item #1**	**...**	**WM-Bus Device Filter Item #N**
	8 Bit	80 Bits	...	80 Bits
	$0B_h$	see WM-Bus Device Filter Item		
		0 <= N <= 32		
Response	**Msg ID**	**Status**		
	8 Bit	8 Bit		
	$0C_h$	see Status Codes		

Get WM-Bus Device Filter Item List

This message is used to read out the current WM-Bus Filter configuration.

HCI Message					
Request	**Msg ID**	**Payload**			
	8 Bit	none			
	$0D_h$				
Response	**Msg ID**	**Status**	**WM-Bus Device Filter Item #1**	**...**	**WM-Bus Device Filter Item #N**
	8 Bit	8 Bit	80 Bits	...	80 Bits
	$0E_h$	see Status Codes	see WM-Bus Device Filter Item		
			0 <= N <= 32		

WM-Bus Device Filter Item

The next figure outlines the detailed format of a single WM-Bus Device Filter Item

WM-Bus Device Filter Item					
WM-Bus Address Fields[1)]				**Address Field Mask**	**Filter Group ID**
Manufacturer ID	**Device ID**	**Version**	**Type**		
16 Bit	32 Bit	8 Bit	8 Bit	8 Bit	8 Bit

Note
1) The byte ordering of multi byte fields is the same as in the Wireless M-Bus packets transmitted over the air.

- **WM-Bus Address Fields**
  A sequence of 8 bytes in total which are transmitted in the header of each Wireless M-Bus packet.

- **Address Field Mask**
  This mask defines which of the single WM-Bus Address Field Bytes is used for comparison with every received WM-Bus packet.
  Bit 0 = Type
  Bit 1 = Version
  Bit 2 .. 5 = Device ID Bytes
  Bit 6 .. 7 = Manufacturer IDBytes

- **Filter Group ID**
  This element is only used to group several filter items.
  Note: The value 255 (FF_h) is reserved and means that this filter item should be applied independent of the configuration of an Calender Event Item.

Back to WM-Bus Range Extender Servcies

Configurable Range Extender Options

The following messages can be used to read and write further Range Extender options.

Set Range Extender Options

This message is used to set the Range Extender option bits.

HCI Message		
Request	**Msg ID**	**Option Bits**
	8 Bit	32 Bits
	41_h	see Range Extender Options
Response	**Msg ID**	**Status**
	8 Bit	8 Bit
	42_h	see Status Codes

Get Range Extender Options

This message is used to read out the Range Extender option bits

HCI Message			
Request	**Msg ID**	**Payload**	
	8 Bit	none	
	43_h		
Response	**Msg ID**	**Status**	**Option Bits**
	8 Bit	8 Bit	32 Bits
	44_h	see Status Codes	see Range Extender Options

Range Extender Option Bits

This resource provides some extra configuration parameters which control the behaviour of the Wireless M-Bus Range Extender.

Range Extender - Extras
Option Bits
32 Bit, LSB first

- **Options Bits**
  This field includes several configuration bits:

  Bit 0 : Duplicate WM-Bus Packet Filter:
  0 = disabled
  1 = enabled

  Bit 1 : Duplicate WM-Bus Packet Filter with CRC :
  0 = Verification of WM-Bus Header bytes only
  1 = Verification of WM-Bus Header bytes and additional Packet CRC
  Note: Bit 0 must be enabled too

  Bit 2 - 3 : reserved for future

  Bit 4 : LED usage for status signalling:
  0 = disabled
  1 = enabled

  Bit 5 : WM-Bus Messages with RSSI (Firmware Version 1.1)
  0 = disabled
  1 = enabled
  Note: WM-Bus Message including RSSI will be uploaded on dedicated LoRaWAN Ports.

  Bit 6 - 31 : reserved for future

Back to Accessible Resources

Configurable LoRaWAN Device EUI

The following messages are used to read and write the LoRaWAN Device EUI.

Set LoRaWAN Device EUI

This message is used to write a new LoRaWAN Device EUI

Note: A device must be re-activated if one of the LoRaWAN connectivity parameters has changed.

HCI Message		
Request	**Msg ID**	**LoRaWAN Device EUI**
	8 Bit	64 Bits
	11_h	
Response	**Msg ID**	**Status**
	8 Bit	8 Bit
	12_h	see Status Codes

Get LoRaWAN Device EUI

This message is used to read out the current LoRaWAN Device EUI.

HCI Message			
Request	**Msg ID**	**Payload**	
	8 Bit	none	
	13_h		
Response	**Msg ID**	**Status**	**LoRaWAN Device EUI**
	8 Bit	8 Bit	64 Bit
	14_h	see Status Codes	

Back to WM-Bus Range Extender Servcies

Configurable LoRaWAN OTAA Settings

The following messages are used to read and write the LoRaWAN OTAA parameters.

Set LoRaWAN OTAA Configuration

This message is used to write new parameters for LoRaWAN Over The Air Activation.

Note: A device must be re-activated if one of the LoRaWAN connectivity parameters has changed.

HCI Message			
Request	**Msg ID**	**Application EUI**	**Device Key**
	8 Bit	64 Bit	128 Bit
	15_h		
Response	**Msg ID**	**Status**	
	8 Bit	8 Bit	
	16_h	see Status Codes	

Get LoRaWAN OTAA Configuration

This message is used to read out the current LoRaWAN OTAA configuration.

HCI Message			
Request	**Msg ID**	**Payload**	
	8 Bit	none	
	17_h		
Response	**Msg ID**	**Status**	**Application EUI**
	8 Bit	8 Bit	64 Bit
	18_h	see Status Codes	

Note
The Device Key is not readable.

Back to WM-Bus Range Extender Servcies

Configurable LoRaWAN ABP Settings

The following messages are used to read and write the LoRaWAN ABP parameters.

Set LoRaWAN ABP Configuration

This message is used to write new parameters for LoRaWAN Activation by Personalization.

Note: A device must be re-activated if one of the LoRaWAN connectivity parameter has changed.

HCI Message						
Request	**Msg ID**	**Link Address**	**Network Session Key**	**Application Session Key**		
	8 Bit	32 Bit	128 Bit	128 Bit		
	19$_h$					
Response	**Msg ID**	**Status**				
	8 Bit	8 Bit				
	1A$_h$	see Status Codes				

Get LoRaWAN ABP Configuration

This message is used to read out the current LoRaWAN ABP configuration.

HCI Message							
Request	**Msg ID**	**Payload**					
	8 Bit	none					
	1B$_h$						
Response	**Msg ID**	**Status**	**Link Address**				
	8 Bit	8 Bit	32 Bit				
	1C$_h$	see Status Codes					

Note
The Network Session Key and Application Session Key are not readable.

Back to WM-Bus Range Extender Servcies

Configurable LoRaWAN Activation Type

The following messages are used to read and write the LoRaWAN Activation Type.

Set LoRaWAN Activation Type

This message is used to set the next LoRaWAN Activation Type

Note: A device must be re-activated if one of the LoRaWAN connectivity parameter has changed.

HCI Message		
Request	**Msg ID**	**Activation Type**
	8 Bit	8 Bit
	$1D_h$	00_h = ABP 01_h = OTAA
Response	**Msg ID**	**Status**
	8 Bit	8 Bit
	$1E_h$	see Status Codes

Get LoRaWAN Activation Type

This message is used to read out the current LoRaWAN Activation Type.

HCI Message			
Request	**Msg ID**	**Payload**	
	8 Bit	none	
	$1F_h$		
Response	**Msg ID**	**Status**	**Activation Type**
	8 Bit	8 Bit	8 Bit
	20_h	see Status Codes	

Back to WM-Bus Range Extender Servcies

Configurable LoRaWAN Stack Settings

The following messages are used to read and write the general LoRaWAN Stack parameters.

Set LoRaWAN StackConfiguration

HCI Message			
Request	**Msg ID**	**Reserved**	**Stack Configuration**
	8 Bit	16 Bit	6 * 8 Bit
	25_h	can be set to 0	
Response	**Msg ID**	**Status**	
	8 Bit	8 Bit	
	26_h	see Status Codes	

Get LoRaWAN Stack Configuration

This message is used to read out the current LoRaWAN OTAA configuration.

HCI Message							
Request	**Msg ID**	**Payload**					
	8 Bit	none					
	27_h						
Response	**Msg ID**	**Status**	**Reserved**	**Stack Configuration**			
	8 Bit	8 Bit	16 Bit	6 * 8 Bit			
	28_h	see Status Codes					

Stack Configuration

The following table includes the Stack Configuration items.

Stack Configuration				
Options	**Uplink Retries**	**Data Rate**	**Uplink Power Level**	**MAC Command Capacity**
16 Bit	8 Bit	8 Bit	8 Bit, signed	8 Bit

- **Options**
  This field includes several options bits:

  Bit 0 : **Adaptive Data Rate (ADR)**
  0 = off, 1 = on

  Bit 1 : **LoRaWAN Network Type**
  0 = public LoRaWAN, 1 = private LoRaWAN

  Bit 2 : **Duty Cycle Control**

Must be set to "1" to enable the required duty cycle management.

Bit 3 ... Bit 15: reserved, should be set to "0"

- **Tx Retries**
  Maximum number of uplink retries (default: 12)
- **Data Rate**
  Initial data rate (0 .. 5, 0 = SF12, 1 = SF11, ... , 5 = SF7, default : SF12)
- **Uplink Power Level**
  Transmit power level in dbm (-1 dBm .. 13 dBm, default 13 dBm)
- **MAC Command Capacity**
  Maximum number of bytes in uplink packets for MAC commands: (0 - 15, default 15)

Back to WM-Bus Range Extender Servcies

Get Application Status

The following message can be used to retrieve the Application Status.

HCI Message											
Request	**Msg ID**	**Payload**									
	8 Bit	none									
	05_h										
										New for Firmware 1.1	
Respons e	**Msg ID**	**Date and Time (UTC)**	**Firmware Version**	**Last Sync Time**	**Reset Counter$^{1)}$**	**Status**	**WM-Bus Rx Counter$^{2)}$**	**WM-Bus Stored Counter$^{2)}$**	**WM-Bus Tx Counter$^{2)}$**	**Battery Voltage**	**Firmware Type**
	8 Bit	32 Bit, LSB first	16 Bit, Minor version first	32 Bit, LSB first	32 Bit, LSB first	16 Bit, LSB first	32 Bit, LSB first	32 Bit, LSB first	32 Bit, LSB first	16 Bit, LSB first	8 Bit
	06_h	see Date and Time Services	e.g. (07 01 $)_h$ V1.7							Value in mV	00_h = Release 01_h = Field Test Beta XX_h = Reserved

- **Date and Time**
  Contains the current date and time in seconds since 01.01.1970 00:00:00

- **Firmware Version**
  Minor and major firmware version

- **Last Sync Time**
  Contains the time stamp of the latest sychronization via local or air interface

- **Reset Counter$^{1)}$**
  Contains the number of device resets

- **Status**
  This field includes several Status Bits:

  Bit 0 : 1 = LoRaWAN Stack is not activated

  Bit 1 : 1 = Network Time is not synchronized

  Bit 2 : 1 = System Time is not synchronized

  Bit 3 : Reserved

  Bit 4 : 1 = LoRa Configuration is invalid

  Bit 5 : 1 = WM-Bus Device Filter list is empty

  Bit 6 : 1 = Calendar event list is empty

  Bit 7 : 1 = Limited Access, LoRaWAN and WM-Bus radio functionality disabled

  Bit 8 : 1 = Flash Memory full condition detected

  Bit 9 : 1 = Flash Memory CRC error detected

- **WM-Bus Rx Counter$^{2)}$**
  Total received WM-Bus packets before any packet filtering since last counter reset

- **WM-Bus Stored Counter$^{2)}$**
  Number of stored WM-Bus packets after packet filtering

- **WM-Bus Tx Counter**$^{2)}$
  Number of uploaded WM-Bus packets

- **Battery Voltage**
  The battery voltage is measured just before transmitting this status message. The value is returned in Millivolts.

- **Firmware Type**
  This element indicates different types of firmware verson: e.g. official released version or field test beta version.

Info
[1] The Reset Counter is copied to the non-volatile memory earliest 30 seconds after system start.

[2] The WM-Bus packet counters are written into the non-volatile memory earliest 30 seconds after last increment. These counters can be reset by means of an HCI message.

Back to WM-Bus Range Extender Servcies

Application Events

Application Events are used to trigger certain firmware activities. These events can be scheduled by Calendar Events or immediately by means of this HCI messages.

Trigger Application Event

This message can be used to trigger a firmware activity immediately.

HCI Message							
Request	**Msg ID**	**Application Event ID**					
	8 Bit	16 Bit, LSB first					
	31_h	see ApplicationEvents					
Response	**Msg ID**	**Status**					
	8 Bit	8 Bit					
	32_h	see Status Codes					

Application Events

The following table lists all application events

Event Name	Event ID	via Calendar	via HCI Interface	Description
None	00_h	no	no	Invalid event
UI Events				
Show Status	01_h	yes	yes	Output of internal status on LED
Push Button	02_h	yes	yes	Simulates the push button function: performs LoRaWAN Activation per OTAA or ABP or if already activated displays the status on LED
LED Off	03_h	yes	yes	Set LED off
LED Red	04_h	yes	yes	Set LED red color
LED Green	05_h	yes	yes	Set LED green color
LED Yellow	06_h	yes	yes	Set LED yellow color
LED Red Blinking	07_h	yes	yes	Set LED red blinking
LED Green Blining	08_h	yes	yes	Set LED green blinking
LED Yellow Blinking	09_h	yes	yes	Set LED yellow blinking
LoRaWAN Events				
LoRaWAN Activate	20_h	not recommended	yes	Activate LoRaWAN Stack per OTAA or ABP
LoRaWAN Deactivate	21_h	not recommended	yes	Deactivate LoRAWAN Stack
System Events				

Get LoRaWAN Network Time	30_h	yes	yes	Request the date and time via LoRaWAN MAC command. On response the system time will be synchronized.
Send Application Status	31_h	yes	yes	Transmit Application Status via LoRaWAN
Get App Network Time	32_h	yes	yes	Requests the date and time by means of an application message via LoRaWAN. On response the system time will be synchronized.
Erase Flash	33_h	yes	yes	Erases the external flash memory content. Note: This operation can take up to 32 seconds.
WM-Bus Events				
Receive in S-Mode and record	40_h	yes	yes	Enable receiver for Wireless M-Bus S-Mode, received messages will be filtered and stored in non-volatile flash memory.
Receive in C/T-Mode and record	41_h	yes	yes	Enable receiver for Wireless M-Bus C/T-Mode, received messages will be filtered and stored in non-volatile flash memory
Receiver Off	42_h	yes	yes	Disable receiver
Start Upload	43_h	yes	yes	Disable receiver and start upload of stored WM-Bus messages via LoRaWAN
Receive S-Mode and output via HCI	44_h	yes	yes	Enable receiver for Wireless M-Bus S-Mode, received messages will be forwarded via HCI
Received C/T-Mode and output via HCI	45_h	yes	yes	Enable receiver for Wireless M-Bus C/T-Mode, received messages will be forwarded via HCI

Back to WM-Bus Range Extender Servcies

Application Notifications

This message is sent to the connected Host Controller to notify an application event.

Event	Msg ID	Date and Time (UTC)	Notification ID	Parameter (optional)				
	8 Bit	32 Bit, LSB first	8 Bit	8 Bit				
	34_h	see Date and Time Services	see Notification IDs					

Notification IDs

Notification Name	Notification ID	Parameter	Description
LoRaWAN Activation started	01_h	00_h = ABP, 01_h = OTAA	LoRaWAN Activation is active, response from server is outstanding.
LoRaWAN Activation terminated	02_h	00_h = OK, 01_h = failed	The LoRaWAN activation has terminated. On success the device is ready for further uplink & downlink packets.
Network Time Synchronization started	03_h	none	Network time synchronization via LoRaWAN is active, response from server is outstanding
Network Time Synchronization terminated	04_h	00_h = OK, 01_h = failed	The sychronization process has terminated. On success the embedded RTC has been snyChronized to the servers network time, see Application Status.
Application Status Transmission started	05_h	none	The Application Status transmission is active, response from server is outstanding.
Application Status Transmission terminated	06_h	00_h = OK, 01_h = failed	The Application Status transmission has terminated.
WM-Bus Reception started	07_h	00_h = S-Mode 01_h = C-/T-Mode	The Wireless M-Bus reception phase is active.
WM-Bus Reception terminated	08_h	none	The Wireless M-Bus reception phase has stopped.
WM-Bus Recording started	09_h	00_h = S-Mode 01_h = C-/T-Mode	The Wireless M-Bus recording phase is active. Received packets are filtered and stored in NVM.
WM-Bus Recording terminated	$0A_h$	none	The Wireless M-Bus recording phase has stopped.
WM-Bus Packet Upload started	$0B_h$	none	The Wireless M-Bus packet upload procedure is active.
WM-Bus Packet Upload terminated	$0C_h$	none	The Wireless M-Bus packet upload procedure has terminated.
LoRaWAN Activation not started	$0D_h$	none	LoRaWAN Activation not started due to invalid configuration. Please verify the LoRaWAN Device EUI!
LoRaWAN Deactivated	$0E_h$	none	LoRaWAN Stack is deactivated.

Flash Erased	0F$_h$	none	The external flash memory has been erased.

Back to WM-Bus Range Extender Servcies

WM-Bus Packet Notification

This message is sent to the connected Host Controller to notify a received WM-Bus packet.

Event	Msg ID	Date and Time (UTC)	Reserved	WM-Bus Packet				
	8 Bit	32 Bit, LSB first	16 Bit	n * 8 Bit				
	36_h	see Date and Time Services		see WM-Bus Packet Format				

This message will be sent to the Host Controller to notify a received WM-Bus packet in firmware version 1.1 ff.

Event	Msg ID	Date and Time (UTC)	Reserved	RSSI	WM-Bus Packet				
	8 Bit	32 Bit, LSB first	16 Bit	8 Bit, signed	n * 8 Bit				
	$3C_h$	see Date and Time Services		RSSI in dBm	see WM-Bus Packet Format				

WM-Bus Packet Format

The WM-Bus Format used on the local serial interface and the LoRaWAN air inteface looks as follows:

WM-Bus Packet						
Link Layer Header						Further Data
L-Field	C-Field	Man ID Field	Device ID Field	Version	Type	
8 Bit	8 Bit	16 Bit	32 Bit	8 Bit	8 Bit	n * 8 Bit

Note
The WM-Bus Range Extender keeps the content of the original received WM-Bus messages. Only the CRCs for WM-Bus Packet Format A and B are verified and stripped off. For Packet Format B a correction of the L-Field value (Packet Length) by 2 CRC bytes is automatically done.

Back to WM-Bus Range Extender Servcies

Reset WM-Bus Packet Counter

This message can be used to reset the WM-Bus packet counters, which can be read via Application Status.

HCI Message									
Request	**Msg ID**	**Payload**							
	8 Bit	none							
	37_h								
Response	**Msg ID**	**Status**							
	8 Bit	8 Bit							
	38_h	see Status Codes							

Back to WM-Bus Range Extender Servcies

WM-Bus Range Extender Status Codes

The following table lists the possible status codes for this Service Access Point.

Status	Beschreibung
00_h	ok
01_h	error
02_h	command not supported
03_h	wrong parameter
04_h	wrong application mode
05_h	reserved
06_h	application busy, try later
07_h	wrong message length
08_h	NVM write error
09_h	NVM read error (NVM content is invalid)
$0A_h$	command rejected, execution not possible in current application state

Table : WM-Bus Range Extender Status Codes

NVM = none-volatile memory

Back to WM-Bus Range Extender Services

Appendix - Example Code

The following sub chapters present some HCI Protocl example code written in C/C++.

- Example Code - Device Management Messages
- Example Code - Serial Message
- Example Code - SLIP Decoder
- Example Code - SLIP Encoder
- Example Code - SLIP Encoder (Bufferless Version)
- Example Code - CRC16

Example Code - Device Management Messages

DeviceManagement.cpp

```
/**
 * @file        DeviceManagement.cpp
 *
 * @brief       Implementation of Device Management services
 *
 * @note        This example code is provided by IMST GmbH on an "AS IS"
basis without any warranties.
 */

#include "SerialMessage.h"
#include "SlipEncoder.h"

enum SapIDs
{
 DeviceMgmt_ID = 0x01
};

enum MessageIDs
{
 Ping_Req    = 0x01,
 Ping_Rsp    = 0x02,
 GetDeviceInfo_Req  = 0x03,
 GetDeviceInfo_Rsp  = 0x04,
 GetFirmwareInfo_Req = 0x05,
 GetFirmwareInfo_Rsp = 0x06
}

void
Ping()
{
 SerialMessage msg( DeviceMgmt_ID, Ping_Req );

 SendMsg( msg );
}
```

```
void
GetDeviceInformation()
{
 SerialMessage msg( DeviceMgmt_ID, GetDeviceInfo_Req );

 SendMsg( msg );
}



void
GetFirmwareInformation()
{
 SerialMessage msg( DeviceMgmt_ID, GetFirmwareInfo_Req );

 SendMsg( msg )
}

void
SendMsg( SerialMessage& msg )
{
    msg.Append_CRC16();

    QByteArray outputData;

    // send SLIP encoded stream via serial port
    SerialPort.write( SlipEncoder::Encode( outputData, msg ) );
}

void
ProcessReceivedMsg( SerialMessage& serialMsg )
{
 // CRC ok ?
 if ( serialMsg.CheckCRC16() )
 {
        // remove trailing CRC16 for correct message/payload length
        serialMsg.RemoveCRC16();


  uint8_t sapID = serialMsg.GetSapID();

  switch (  sapID )
  {
   case DeviceMgmt_ID:
    ProcessDeviceMgmtMsg( serialMsg );
    break;

   //... add furher SAPs here
  }
 }
}


void
```

```
ProcessDeviceMgmtMsg( const SerialMessage& response )
{
 uint8_t msgID = response.GetMsgID();


 switch( msgID )
 {
  case Ping_Rsp:
   ProcessPingResponse( response );
   break;


  case GetDeviceInfo_Rsp:
   ProcessDeviceInfoResponse( response );
   break;


  case GetFirmwareInfo_Rsp:
   ProcessFirmwareInfoResponse( response );
   break;


  // ... add furher MsgIDs here
 }
}

void
ProcessPingResponse( const SerialMessage& reponse )
{
 // ... notify application about successful HCI link
}
void
ProcessDeviceInfoResponse( const SerialMessage& response )
{
    enum ResponseFormat
    {
        Status_Index    = 0,
        ModuleType_Index   = 1,
        ModuleID_Index   = 2,
        MinSize    = 6
    };

    enum Status
    {
        Ok                  = 0
    };

    if ( response.GetPayloadLength() >= MinSize )
    {
        // verify positiv status code
        if ( response.GetPayload_U8( Status_Index ) == Ok )
        {
            uint8_t moduleType = response.GetPayload_U8(
ModuleType_Index );
```

```
            uint32_t moduleID  = response.GetPayload_U32( ModuleID_Index
);

            // add code to pass result e.g. via JSON to application
        }
    }
}

void
ProcessFirmwareInfoResponse( const SerialMessage& response )
{
    enum ResponseFormat
    {
        Status_Index    = 0,
  MinorVersion_Index  = 1,
  MajorVersion_Index  = 2,
  BuildCount_Index  = 3,
  BuildDate_Index  = 5,
  BuildDate_Size   = 10,
  FirmwareName_Index  = ( BuildDate_Index + BuildDate_Size ),
        MinSize    = FirmwareName_Index
    };

    enum Status
    {
        Ok                  = 0
    };

    if ( response.GetPayloadLength() >= MinSize )
    {
        // verify positiv status code
        if ( response.GetPayload_U8( Status_Index ) == Ok )
        {
            uint8_t  minorVersion   =   response.GetPayload_U8(
MinorVersion_Index );
            uint8_t  majorVersion   =   response.GetPayload_U8(
MajorVersion_Index );
            uint16_t  buildCount    =   response.GetPayload_U16(
BuildCount_Index );
            QByteArray buildDate    =   response.GetPayload(
BuildDate_Index, BuildDate_Size );
            QByteArray name         =   response.GetPayload(
FirmwareName_Index );


            // add code to pass result e.g. via JSON to application
        }
        }
    }
```

Example Code - Serial Message

SerialMessage.h

```
#ifndef __SerialMessage_H__
#define __SerialMessage_H__

/**
 * @file    SerialMessage.h
 *
 * @brief   Declaration of class SerialMessage
 *
 * @note    This example code is provided by IMST GmbH on an "AS IS"
basis without any warranties.
 */

#include <QByteArray>
#include <stdint.h>

/**
 * @brief   The class SerialMessage extends QByteArray with following
functionalities
 *          - access to dedicated message fields: header fields, payload
 *          - CRC16 calculation and checking
 *
 * @note    QByteArray provides basic array functionality and memory
management for byte arrays
 *
 */

class SerialMessage : public QByteArray
{
public:


    // serial message format
    enum MessageFormat
    {
        SapID_Index =   0,
        MsgID_Index =   1,
        HeaderSize  =   2
    };

    enum
    {
        InvalidSapID    =   0xFF,
        InvalidMsgID    =   0xFF,
        InvalidLength   =   -1
    };

    /**
 * @brief   class constructor
 */


            SerialMessage();

            SerialMessage( uint8_t sapID, uint8_t msgID );

/**
 * @brief   check CRC16
 *
 * @return  true  - CRC16 ok
 *          false - CRC16 error
 */

bool        CheckCRC16() const;

/**
 * @brief   remove trailing CRC16
 */

void        RemoveCRC16();

/**
 * @return  service access point identifier
 */

uint8_t     GetSapID() const;

/**
 * @return  message identifier
 */

uint8_t     GetMsgID() const;

/**
 * @return  payload length
 */

int         GetPayloadLength() const;

/**
 * @return  U8 value from payload field
 *
 * @param   index   index to payload field
 */

uint8_t     GetPayload_U8( int index ) const;

/**
 * @return  U16 value from payload field (LSB first)
 *
 * @param   index   index to payload
 */
```

```
uint16_t    GetPayload_U16( int index ) const;

/**
 * @return  U32 value from payload field (LSB first)
 *
 * @param   index   index to payload field
 */

uint32_t    GetPayload_U32( int index ) const;

/**
 * @return  U64 value from payload field (LSB first)
 *
 * @param   index   index to payload field
 */

uint64_t    GetPayload_U64( int index ) const;

/**
 * @return  array of bytes from payload field
 *
 * @param   index   index to payload field
 */

QByteArray  GetPayload(int index , int size = -1 ) const;

 /**
  * @brief   init request for transmission
  *
  * @param   sapID   service accesspoint identifier
  * @param   msgID   message identifier
  */

void        InitRequest( uint8_t sapID, uint8_t msgID );

 /**
  * @brief   append U8 value
  *
  * @param   8 Bit value
  *
  * @return  number of appended bytes (1)
  */

int         Append( uint8_t value );

/**
  * @brief   append U16 value (LSB first)
  *
  * @param   16 Bit value
  *
  * @return  number of appended bytes (2)
  */
```

```
int         Append( uint16_t value );

/**
 * @brief   append U32 value (LSB first)
 *
 * @param   32 Bit value
 *
 * @return  number of appended bytes (4)
 */

int         Append( uint32_t value );

/**
 * @brief   append U64 value (LSB first)
 *
 * @param   64 Bit value
 *
 * @return  number of appended bytes (8)
 */

int         Append( uint64_t value );

/**
 * @brief   calculate and append CRC16 for message transmission
 *
 * @return  number of bytes appended (2)
 */

int         Append_CRC16();
```

```
};

#endif // __SerialMessage_H__
```

SerialMessage.cpp

```
/**
 * @file    SerialMessage.cpp
 *
 * @brief   Implementation of class SerialMessage
 *
 * @note    This example code is provided by IMST GmbH on an "AS IS"
basis without any warranties.
 */

#include "SerialMessage.h"
#include "CRC16.h"

SerialMessage::SerialMessage()
{
}

SerialMessage::SerialMessage( uint8_t sapID, uint8_t msgID )
{
    InitRequest( sapID, msgID );
}

bool
SerialMessage::CheckCRC16() const
{
    CRC16   crc16;

    // get reference to base class
    const QByteArray& data = *this;

    return crc16.Check( data );
}

void
SerialMessage::RemoveCRC16()
{
    if ( count() >= ( HeaderSize + (int)sizeof( uint16_t ) ) )
    {
        // remove trailing crc bytes
        chop( 2 );
    }
}

uint8_t
SerialMessage::GetSapID() const
{
    if ( count() >= ( HeaderSize ) )
    {
        return (uint8_t)at( SapID_Index );
    }
    return InvalidSapID;
}

uint8_t
SerialMessage::GetMsgID() const
{
    if ( count() >= ( HeaderSize ) )
    {
        return (uint8_t)at( MsgID_Index );
    }
    return InvalidMsgID;
}

int
SerialMessage::GetPayloadLength() const
{
    if ( count() >= ( HeaderSize ) )
    {
        return ( count() - ( HeaderSize ) );
    }
    return InvalidLength;
}

uint8_t
SerialMessage::GetPayload_U8( int index ) const
{
    if ( count() >= ( HeaderSize + index + 1 ) )
    {
        return (uint8_t)at( HeaderSize + index );
    }
    return 0;
}

uint16_t
SerialMessage::GetPayload_U16( int index ) const
{
    if ( count() >= ( HeaderSize + index + 2 ) )
    {
        return (uint16_t)( (uint8_t)at( HeaderSize + 0 + index ) <<  0 )
|
               (uint16_t)( (uint8_t)at( HeaderSize + 1 + index ) <<  8
);
    }
    return 0;
}

uint32_t
SerialMessage::GetPayload_U32( int index ) const
{
    if ( count() >= ( HeaderSize + index + 4 ) )
    {
        return (uint32_t)( (uint8_t)at( HeaderSize + 0 + index ) <<  0 )
|
               (uint32_t)( (uint8_t)at( HeaderSize + 1 + index ) <<  8 )
|
               (uint32_t)( (uint8_t)at( HeaderSize + 2 + index ) << 16 )
|
               (uint32_t)( (uint8_t)at( HeaderSize + 3 + index ) << 24
);
    }
    return 0;
}

uint64_t
SerialMessage::GetPayload_U64( int index ) const
{
    if ( count() >= ( HeaderSize + index + 8 ) )
    {
        uint32_t lo = GetPayload_U32( index );
        uint32_t hi = GetPayload_U32( index + 4 );

        return (uint64_t)( ( (uint64_t)hi << 32 ) |  lo );

    }
    return 0;
}

QByteArray
SerialMessage::GetPayload( int index, int size ) const
{
    if ( size != -1 )
    {
        if ( count() >= ( HeaderSize + index + size ) )
        {
            // return remaining payload
            return mid( HeaderSize + index, size );
        }
    }
    else
    {
        // return remaining part of payload
        if ( count() > ( HeaderSize + index ) )
        {
            // return remaining payload
            return mid( HeaderSize + index, size );
        }
    }
    // return empty array
    return QByteArray();
}

void
SerialMessage::InitRequest( uint8_t sapID, uint8_t msgID )
{
    // clear buffer for init
    clear();

    // attach HCI header
    append( sapID );
    append( msgID );
}

int
SerialMessage::Append( uint8_t value )
{
    // append single byte
    append( (uint8_t)( value ) );

    // 1 byte appended
    return 1;
}

int
SerialMessage::Append( uint16_t value )
{
    // LSB first
    append( (uint8_t)( value ) );
    append( (uint8_t)( value >> 8 ) );

    // 2 bytes appended
    return 2;
}

int
SerialMessage::Append( uint32_t value )
{
    // LSB first
    append( (uint8_t)( value ) );
    append( (uint8_t)( value >> 8 ) );
    append( (uint8_t)( value >> 16 ) );
    append( (uint8_t)( value >> 24 ) );


    // 4 bytes appended
    return 4;

}

int
SerialMessage::Append( uint64_t value )
{
    // LSB first
    append( (uint8_t)( value ) );
    append( (uint8_t)( value >> 8 ) );
    append( (uint8_t)( value >> 16 ) );
    append( (uint8_t)( value >> 24 ) );
    append( (uint8_t)( value >> 32 ) );
    append( (uint8_t)( value >> 40 ) );
    append( (uint8_t)( value >> 48 ) );
    append( (uint8_t)( value >> 56 ) );


    // 8 bytes appended
    return 8;
}

int
SerialMessage::Append_CRC16()
{
    CRC16   crc16;

    // get reference to base class
    const QByteArray& data = *this;
    // append one's complement of crc
    return Append( (uint16_t)~crc16.Calc( data ) );
}
```

Example Code - SLIP Decoder

SlipDecoder.h

```
#ifndef __Slip_Decoder_H__
#define __Slip_Decoder_H__

/**
 * @file    SlipDecoder.h
 *
 * @brief   Declaration of class SlipDecoder
 *
 * @note    This example code is provided by IMST GmbH on an "AS IS"
basis without any warranties.
 */

#include <QByteArray>
#include <QObject>

/**
 * @brief   The SlipDecoder class decodes SLIP encoded byte streams.
 *
 * @note    This class is derived from Q_OBJECT for Qt's signal/slot
mechanism.
 */

class SlipDecoder: public QObject
{
    Q_OBJECT

public:


    /**
     * @brief   class constructor
     */

                SlipDecoder();

    /**
     * @brief   reset decoder
     */

    void        Reset();

    /**
     * @brief   decode encoded SLIP stream
     *
     * @param   output      decoded frame
     * @param   input       SLIP encoded byte stream
     *
     * @note    on signal "OnFrameReady" the decoded SLIP frame is ready
the output array
     */

    void        Decode( QByteArray& output, const QByteArray& input );

signals:

    /**
     * @brief   notification that a SLIP frame has been decoded
successfully
     *          and is ready for further processing
     */

    void        OnSlipDecoder_FrameReady();

private:

    /**
     * standard SLIP frame characters
     */

    enum FrameCharacters
    {
        Begin   =   0xC0,
        End     =   0xC0,
        Esc     =   0xDB,
        EscEnd  =   0xDC,
        EscEsc  =   0xDD
    };

    /**
     * decoder states
     */


    enum DecoderState
    {
        Initial = 0,
        InFrame,
        EscState
    };

    //<! decoder state
    DecoderState        State;
};
#endif // __Slip_Decoder_H__
```

SlipDecoder.cpp

```
/**
 * @file        SlipDecoder.cpp
 *
 * @brief       Implementation of class SlipDecoder.
 *
 * @note        This example code is provided by IMST GmbH on an "AS IS"
basis without any warranties.
 */

#include "SlipDecoder.h"
SlipDecoder::SlipDecoder()
           : State( SlipDecoder::Initial )
{
}

void
SlipDecoder::Reset()
{
    State = SlipDecoder::Initial;
}

void
SlipDecoder::Decode( QByteArray& output, const QByteArray& input )
{
    for ( int index = 0; index < input.count(); index++ )
    {
        uint8_t byte = (uint8_t)input.at( index );


        switch ( State )
        {
            case SlipDecoder::Initial:
                // begin of SLIP frame ?
                if ( byte == SlipDecoder::Begin )
                {
                    // reset output buffer
                    output.clear();

                    State = SlipDecoder::InFrame;
                }
                break;

            case SlipDecoder::InFrame:
                // end of SLIP frame ?
                if ( byte == SlipDecoder::End )
                {
                    State = SlipDecoder::Initial;

                    // notify client that SLIP frame is ready in output
buffer
                    emit OnSlipDecoder_FrameReady();
                }
                // SLIP esc ?
                else if ( byte == SlipDecoder::Esc )
                {
                    State = SlipDecoder::EscState;
                }
                else
                {
                    // default case
                    output.append( byte );
                }
                break;


           case  SlipDecoder::EscState:
                // end of escape state ?
                if ( byte == SlipDecoder::EscEnd )
                {
                    output.append( SlipDecoder::End );

                    State = InFrame;
                }
                // end of escape state ?
                else if ( byte == SlipDecoder::EscEsc )
                {
                    output.append( SlipDecoder::Esc );

                    State = SlipDecoder::InFrame;
                }
                else // error
                {
                    // abort frame reception -> return to initial state
                    State = SlipDecoder::Initial;
                }
                break;


        } // switch ( State )
        } // for...
    }
```

Example Code - SLIP Encoder

SlipEncoder.h

```
#ifndef __Slip_Encoder_H__
#define __Slip_Encoder_H__

/**
 * @file    SlipEncoder.h
 *
 * @brief   Declaration of class SlipEncoder
 *
 * @note    This example code is provided by IMST GmbH on an "AS IS"
basis without any warranties.
 */

#include <QByteArray>

/**
 * @brief   The SlipEncoder class encodes a byte stream into a SLIP
encoded byte stream
 */

class SlipEncoder
{
public:

    /**
     * @brief   encode byte stream
     *
     * @param   output      encoded SLIP stream
     * @param   input       bytes to encode
     *
     * @return  output      updated output buffer with SLIP encoded byte
stream
     */

    static QByteArray&  Encode( QByteArray& output, const QByteArray&
input );

private:

    /**
     * standard SLIP frame characters
     */

    enum FrameCharacters
    {
        Begin   =   0xC0,
        End     =   0xC0,
        Esc     =   0xDB,
        EscEnd  =   0xDC,
        EscEsc  =   0xDD
    };
};
#endif // __Slip_Encoder_H__
```

SlipEncoder.cpp

```
/**
 * @file        SlipEncoder.cpp
 *
 * @brief       Implementation of class SlipEncoder.
 *
 * @note        This example code is provided by IMST GmbH on an "AS IS"
basis without any warranties.
 */

#include "SlipEncoder.h"

QByteArray&
SlipEncoder::Encode( QByteArray& output, const QByteArray& input )
{
    output.append( SlipEncoder::Begin );

    for ( int index = 0; index < input.count(); index++ )
    {
        uint8_t byte = (uint8_t)input.at( index );

        switch ( byte )
        {
            case  SlipEncoder::End: // same as SLIP Begin !
                    output.append( SlipEncoder::Esc );
                    output.append( SlipEncoder::EscEnd );
                    break;

            case  SlipEncoder::Esc:
                    output.append( SlipEncoder::Esc );
                    output.append( SlipEncoder::EscEsc );
                    break;

            default:
                    output.append( byte );
                    break;
        }
    }

    output.append( SlipEncoder::End );

    return output;
}
```

Example Code - SLIP Encoder (Bufferless Version)

SlipEncoder.h

```
#ifndef __Slip_Encoder_H__
#define __Slip_Encoder_H__

/**
 * @file    SlipEncoder.h
 *
 * @brief   Declaration of class SlipEncoder
 *
 * @note    This example code is provided by IMST GmbH on an "AS IS"
basis without any warranties.
 */

#include <QByteArray>

/**
 * @brief   The SlipEncoder class encodes a byte stream into a SLIP
encoded byte stream without the
 *          need for an additional encoding buffer
 */

class SlipEncoder
{
public:

    /**
     * @brief   class constructor
     */
                        SlipEncoder();

    /**
     * @brief   prepare encoder for SLIP message
     *
     * @param   input               array with message to be encoded
     * @param   numWakeupChars      optional number of wakeup chars
which s
     *                              should be transmitted first
     */

    bool                SetInput( const QByteArray* input, uint16_t
numWakeupChars = 0 );

    /**
     * @brief   return a single SLIP encoded byte
     *
     * @return  if >= 0 -> SLIP encoded byte in lower 8 bit
     *          else       last byte has already been encoded -> UART
transmitter can be configured
     *                     for final TX SHIFT REGISTER empty interrupt
now
     *
     * @note    don't forget to call OnCompleteIndication() to cleanup
encoding state for next message
     */

    int16_t             GetEncodedByte();

    /**
     * @brief   handle completion event which should occur after
     *          the last byte has left the UART TX SHIFT register
     */


    void                OnCompleteIndication();

private:

    /**
     * standard SLIP frame characters
     */
    enum FrameCharacters
    {
        Begin   =   0xC0,
        End     =   0xC0,
        Esc     =   0xDB,
        EscEnd  =   0xDC,
        EscEsc  =   0xDD
    };

    enum EncoderState
    {
        Idle    =   0,
        Wakeup,
        Start,
        InFrame,
        EscEndState,
        EscEscState,
        WaitForCompletion,
        EndState
    };

    EncoderState        State;
    const QByteArray*   Input;
    int                 Index;
    int                 NumWakeupChars;
```

```
};

#endif // __Slip_Encoder_H__
```

SlipEncoder.cpp

```
/**
 * @file        SlipEncoder.cpp
 *
 * @brief       Implementation of class SlipEncoder.
 *
 * @note        This example code is provided by IMST GmbH on an "AS IS"
basis without any warranties.
 */

#include "SlipEncoder.h"

SlipEncoder::SlipEncoder()
           : State ( SlipEncoder::Idle )
           , Input ( 0 )
           , Index ( 0 )
           , NumWakeupChars ( 0 )

{
}

/**
 * @brief   prepare encoder for SLIP message
 *
 * @param   input                 array with message to be encoded
 * @param   numWakeupChars        optional number of wakeup chars which s
 *                                should be transmitted first
 */

bool
SlipEncoder::SetInput( const QByteArray* input, uint16_t numWakeupChars
)
{
    if ( ( State == SlipEncoder::Idle ) && ( input != nullptr ) )
    {
        Input           = input;
        Index           = 0;
        NumWakeupChars  = numWakeupChars;


        if ( NumWakeupChars > 0 )
        {
            State = SlipEncoder::Wakeup;
        }
        else
        {
            State = SlipEncoder::Start;
        }
        return true;
    }
    return false;
}

/**
 * @brief   return a single SLIP encoded byte
 *
 * @return  if >= 0 -> SLIP encoded byte in lower 8 bit
 *          else       last byte has already been encoded -> UART
transmitter can be configured
 *                     for final TX SHIFT REGISTER empty interrupt now
 *
 * @note    don't forget to call OnCompleteIndication() to cleanup
encoding state for next message
 */

int16_t
SlipEncoder::GetEncodedByte()
{
    switch ( State )
    {
        // send wakeup chars first, so that PLL and baudrate generator
on peer device can settle
        case SlipEncoder::Wakeup:


            if ( 0 >= --NumWakeupChars )
            {
                State = Start;
            }
            return SlipEncoder::End;

        // start of frame --> send SLIP_END
        case SlipEncoder::Start:


            State = SlipEncoder::InFrame;
            return SlipEncoder::End;

        // second step of SLIP_END coding --> send SLIP_ESC_END
        case SlipEncoder::EscEndState:

            State = SlipEncoder::InFrame;
            return SlipEncoder::EscEnd;

        // second step of SLIP_ESC coding --> send SLIP_ESC_ESC
        case SlipEncoder::EscEscState:

            State = SlipEncoder::InFrame;
            return SlipEncoder::EscEsc;

        // last byte transmitted --> return EOF so that UART can be
```

```
disabled
        case SlipEncoder::WaitForCompletion:

            State = SlipEncoder::EndState;  // wait for final UART TX
SHIFT Register empty interrupt!
            return -1;

        // normal coding
        case    SlipEncoder::InFrame:
            {
                // eof ?
                if ( Input->count() <= Index )
                {
                    // end of frame --> send terminating SLIP_END
                    State = SlipEncoder::WaitForCompletion;
                    return SlipEncoder::End;
                }

                // get next txByte
                uint8_t txByte = Input->at( Index++ );

                // special character --> send SLIP_ESC
                if ( txByte == SlipEncoder::End )
                {
                    State = SlipEncoder::EscEndState;
                    return SlipEncoder::Esc;
                }
                if ( txByte == SlipEncoder::Esc )
                {
                    State = SlipEncoder::EscEscState;
                    return SlipEncoder::Esc;
                }

                // normal character --> send it
                return (int16_t)txByte;
            }


        case Idle:
        case EndState:
        default:

            // return EOF to disable driver
            return -1;
    }
}

/**
 * @brief   handle completion event which should occur after
 *          the last byte has left the UART TX SHIFT register
 */

void
SlipEncoder::OnCompleteIndication()
```

```
{
    if ( State == SlipEncoder::EndState )
    {
        Input = 0;
        Index = 0;
        State = Idle;
    }
}
```

Example Code - CRC16

CRC16.h

```
#ifndef __CRC16_H__
#define __CRC16_H__

/**
 * @file    CRC16.h
 *
 * @brief   Declaration of class CRC16
 *
 * @note    This example code is provided by IMST GmbH on an "AS IS"
basis without any warranties.
 */

#include <QByteArray>
#include <stdint.h>

/**
 * @brief   The CRC16 class provides methods for CRC calculation and
checking. The implemented CRC uses the well known
 *          16 Bit CCITT Polynom. For performance reason a lookup-table
is used which was generated by means of this polynom.
 */

class CRC16
{
public:

    enum
    {
        Init_Value  =  0xFFFF,    //!< initial value for CRC algorithem
        Good_Value  =  0x0F47,    //!< constant compare value for check
        Polynom     =  0x8408     //!< 16 Bit CRC CCITT Generator
Polynom, used for table generation
    };

    /**
      * @brief  class constructor
      *
      * @param  initValue   initial CRC16 value
      */
                CRC16( uint16_t initValue = CRC16::Init_Value );
    /**
```

```
     * @brief   calculate CRC16
     *
     * @param   data  input data
     *
     * @return  crc16
     */


    uint16_t    Calc( const QByteArray& data );

    /**
     * @brief   calculate and check CRC16
     *
     * @param   data  input data
     *
     * @return  true  - CRC16 ok
     *          false - CRC16 error
     */
    bool        Check( const QByteArray& data );
private:


    //<!  crc value
    uint16_t                CRC;
    //<! static lookup table for fast calculation
```

```
 static const uint16_t   Table[];
};
#endif // CRC16_H
```

CRC16.cpp

```
/**
 * @file    CRC16.cpp
 *
 * @brief   Implementation of class CRC16
 *
 * @note    This example code is provided by IMST GmbH on an "AS IS"
basis without any warranties.
 */

#include "CRC16.h"
const uint16_t
CRC16::Table[] =
{
    0x0000, 0x1189, 0x2312, 0x329B, 0x4624, 0x57AD, 0x6536, 0x74BF,
    0x8C48, 0x9DC1, 0xAF5A, 0xBED3, 0xCA6C, 0xDBE5, 0xE97E, 0xF8F7,
    0x1081, 0x0108, 0x3393, 0x221A, 0x56A5, 0x472C, 0x75B7, 0x643E,
    0x9CC9, 0x8D40, 0xBFDB, 0xAE52, 0xDAED, 0xCB64, 0xF9FF, 0xE876,
    0x2102, 0x308B, 0x0210, 0x1399, 0x6726, 0x76AF, 0x4434, 0x55BD,
    0xAD4A, 0xBCC3, 0x8E58, 0x9FD1, 0xEB6E, 0xFAE7, 0xC87C, 0xD9F5,
    0x3183, 0x200A, 0x1291, 0x0318, 0x77A7, 0x662E, 0x54B5, 0x453C,
    0xBDCB, 0xAC42, 0x9ED9, 0x8F50, 0xFBEF, 0xEA66, 0xD8FD, 0xC974,
    0x4204, 0x538D, 0x6116, 0x709F, 0x0420, 0x15A9, 0x2732, 0x36BB,
    0xCE4C, 0xDFC5, 0xED5E, 0xFCD7, 0x8868, 0x99E1, 0xAB7A, 0xBAF3,
    0x5285, 0x430C, 0x7197, 0x601E, 0x14A1, 0x0528, 0x37B3, 0x263A,
    0xDECD, 0xCF44, 0xFDDF, 0xEC56, 0x98E9, 0x8960, 0xBBFB, 0xAA72,
    0x6306, 0x728F, 0x4014, 0x519D, 0x2522, 0x34AB, 0x0630, 0x17B9,
    0xEF4E, 0xFEC7, 0xCC5C, 0xDDD5, 0xA96A, 0xB8E3, 0x8A78, 0x9BF1,
    0x7387, 0x620E, 0x5095, 0x411C, 0x35A3, 0x242A, 0x16B1, 0x0738,
    0xFFCF, 0xEE46, 0xDCDD, 0xCD54, 0xB9EB, 0xA862, 0x9AF9, 0x8B70,
    0x8408, 0x9581, 0xA71A, 0xB693, 0xC22C, 0xD3A5, 0xE13E, 0xF0B7,
    0x0840, 0x19C9, 0x2B52, 0x3ADB, 0x4E64, 0x5FED, 0x6D76, 0x7CFF,
    0x9489, 0x8500, 0xB79B, 0xA612, 0xD2AD, 0xC324, 0xF1BF, 0xE036,
    0x18C1, 0x0948, 0x3BD3, 0x2A5A, 0x5EE5, 0x4F6C, 0x7DF7, 0x6C7E,
    0xA50A, 0xB483, 0x8618, 0x9791, 0xE32E, 0xF2A7, 0xC03C, 0xD1B5,
    0x2942, 0x38CB, 0x0A50, 0x1BD9, 0x6F66, 0x7EEF, 0x4C74, 0x5DFD,
    0xB58B, 0xA402, 0x9699, 0x8710, 0xF3AF, 0xE226, 0xD0BD, 0xC134,
    0x39C3, 0x284A, 0x1AD1, 0x0B58, 0x7FE7, 0x6E6E, 0x5CF5, 0x4D7C,
    0xC60C, 0xD785, 0xE51E, 0xF497, 0x8028, 0x91A1, 0xA33A, 0xB2B3,
    0x4A44, 0x5BCD, 0x6956, 0x78DF, 0x0C60, 0x1DE9, 0x2F72, 0x3EFB,
    0xD68D, 0xC704, 0xF59F, 0xE416, 0x90A9, 0x8120, 0xB3BB, 0xA232,
    0x5AC5, 0x4B4C, 0x79D7, 0x685E, 0x1CE1, 0x0D68, 0x3FF3, 0x2E7A,
    0xE70E, 0xF687, 0xC41C, 0xD595, 0xA12A, 0xB0A3, 0x8238, 0x93B1,
    0x6B46, 0x7ACF, 0x4854, 0x59DD, 0x2D62, 0x3CEB, 0x0E70, 0x1FF9,
    0xF78F, 0xE606, 0xD49D, 0xC514, 0xB1AB, 0xA022, 0x92B9, 0x8330,
    0x7BC7, 0x6A4E, 0x58D5, 0x495C, 0x3DE3, 0x2C6A, 0x1EF1, 0x0F78,
};
```

```
CRC16::CRC16( uint16_t initValue )
     : CRC  ( initValue )
{
}

uint16_t
CRC16::Calc( const QByteArray& data )
{
    int length = data.count();
    int index  = 0;

    // iterate over all bytes
    while ( length-- )
    {
        // calc new crc
        CRC = ( CRC >> 8 ) ^ Table[ ( CRC ^ data.at( index++ ) ) &
0x00FF ];
    }

    // return result
    return CRC;
}

bool
CRC16::Check( const QByteArray& data )
{
    // get 1's compelemnt
    uint16_t crc = ~Calc( data );
```

```
      // compare it with constant good value
   return (bool) ( crc == CRC16::Good_Value );
  }
```