Freescale Medical Connectivity Library
API Reference Manual

Document Number: MEDCONLIBAPIRM
Rev. 4
05/2012

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064, Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 10 5879 8000
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Number: MEDCONLIBAPIRM
Rev. 4
05/2012

Revision History

To provide the most up-to-date information, the revision of our documents on the World Wide Web will be the most current. Your printed copy may be an earlier revision. To verify you have the latest information available, refer to:

http://www.freescale.com

The following revision history table summarizes changes contained in this document.

Revision Number	Revision Date	Description of Changes
Rev. 1	10/2009	Initial release.
Rev. 2	07/2010	Updated Reference Material section.
Rev. 3	07/2011	Minor editorial changes
Rev. 4	05/2012	Added new chapter IEEE 11073 Manager

Chapter 1
Before Beginning

Chapter 2
Medical Connectivity Library API Overview

Chapter 3
Transport Layer API

Chapter 4
Medical Connectivity Library API

Chapter 5
Data Structures

Chapter 6
PHDC Host Class API

Chapter 1
Before Beginning

1.1 About This Book

This book describes the Freescale Medical Connectivity Library API functions. It describes in detail the API functions that can be used by application code to develop various Medical device specializations. Table 1-1 shows the summary of chapters included in this book.

Table 1-1. MEDCONLIBAPIRM Summary

Chapter Title	Description
Before Beginning	This chapter provides the prerequisites of reading this book.
Medical Connectivity Library API Overview	This chapter gives an overview of the API functions and how to use them for developing new medical device specialization applications.
Transport Layer API	This chapter discusses Transport Layer API interfaces.
Medical Connectivity Library API	This chapter discusses Medical Connectivity Library API interfaces.
Data Structures	This chapter discusses the various data structures used in the USB device class layer API functions.

1.2 Reference Material

Use this book in conjunction with:

- *Freescale USB Stack with PHDC Device Users Guide* (document USBUG)
- *Freescale USB Stack with PHDC API Reference Manual* (document USBAPIRM)
- *Medical Connectivity Library Users Guide* (document MEDCONLIBUG)
- IEEE Std 11073-20601TM-2008, Health informatics — Personal health device communication-Part 20601: Application profile — Optimized Exchange Protocol.

1.3 Acronyms and Abbreviations

CFV1	ColdFire V1 (MCF51JM128 CFV1 device is used in this document)
CFV2	ColdFire V2 (MCF52221 and MCF52259 CFV2 devices are used in this document)
DIM	Domain Information Model
IDE	Integrated Development Environment
ISO	International Organization for Standardization
IEEE	The Institute of Electrical and Electronics Engineers
JM60	MC9S08JM60 Device
PHD	Personal Healthcare Device
PHDC	Personal Healthcare Device Class
TIL	Transport Independent Layer
USB	Universal Serial Bus

1.4 Function Listing Format

This is the general format of an entry for a function, compiler intrinsic, or macro.

function_name()

A short description of what function **function_name()** does.

Synopsis

Provides a prototype for function **function_name()**.

```
<return_type> function_name(
  <type_1>  parameter_1,
  <type_2>  parameter_2,
  ...
  <type_n>  parameter_n)
```

Parameters

parameter_1 [in] — Pointer to x

parameter_2 [out] — Handle for y

parameter_n [in/out] — Pointer to z

Parameter passing is categorized as follows:

- *In* — Means the function uses one or more values in the parameter you give it without storing any changes.
- *Out* — Means the function saves one or more values in the parameter you give it. You can examine the saved values to find out useful information about your application.
- *In/out* — Means the function changes one or more values in the parameter you give it and saves the result. You can examine the saved values to find out useful information about your application.

Description

Describes the function **function_name()**. This section also describes any special characteristics or restrictions that might apply:

- function blocks or might block under certain conditions
- function must be started as a task
- function creates a task
- function has pre-conditions that might not be obvious
- function has restrictions or special behavior

Return Value

Specifies any value or values returned by function **function_name()**.

See Also

Lists other functions or data types related to function **function_name()**.

Example

Provides an example (or a reference to an example) that illustrates the use of function **function_name()**.

Chapter 2
Medical Connectivity Library API Overview

2.1 Introduction

The Freescale Medical Connectivity Library API consists of the functions that can be used at the application level. These enable you to implement different medical specializations.

2.2 API Overview

This section describes the list of API functions and their use.

Table 2-1 summarizes the Transport Layer API functions.

Table 2-1. Summary of Transport Layer API Functions

No.	API Function	Description
1	TIL_Initialize()	Initializes Transport Independent Layer
2	TIL_DeInitialize()	De-initializes Transport Independent Layer
3	TIL_StartTransport()	Initializes Shim
4	TIL_StopTransport()	De-Initializes Shim
5	TIL_SendAPDU()	Sends data through Shim
6	TIL_RecvApdu()	Receives data from Shim

Table 2-2 summarizes the Medical Connectivity Library API functions.

Table 2-2. Summary of Medical Connectivity Library API Functions

No.	API Function	Description
1	Ieee11073Initialize()	Initializes Medical Connectivity Library
2	AgentSendAssociationRequest()	Sends Association request packet to the Manager
3	AgentSendAssociationReleaseRequest()	Sends Association release request to the Manager
4	AgentSendMeasurements()	Sends single person measurements
5	AgentSendPersonMeasurements()	Sends multi person measurements
6	AddEntryToObsScanList()	Adds an entry to the Observation Scan list
7	AddEntryToScanRptPerVarList()	Adds an entry to the Scan report per var list
8	UpdatePmSegmentEntry()	Updates PM Segment entry data
9	SendSegmentData()	Sends PM Segment data event to the manager

2.3 Using API

This section describes the flow on how to use various Medical Connectivity Library API functions.

2.3.1 Using the Medical Connectivity Library API

This section describes a sequence to use the Medical Connectivity Library API functions from the application.

2.3.1.1 Initialization Flow

To initialize the driver layer, the class driver must:

1. Call 3.1.1"TIL_Initialize()" to initialize the Transport Independent Layer.
2. Call 3.1.2"Ieee11073Initialize()" to initialize the Medical Connectivity Library and to start transport.

Chapter 3
Transport Layer API

This section discusses the Transport Layer API functions in detail.

3.1 Transport Layer API Function Listings

3.1.1 TIL_Initialize()

Initializes the Transport Independent Layer (TIL).

Synopsis

```
void TIL_Initialize(PTIL pTil))
```

Parameters

pTil [in] — Pointer to TIL

Description

This function initializes TIL with the input pointer.

Return Value

None

3.1.2 TIL_DeInitialize()

De-Initializes the Transport Independent Layer.

Synopsis

```
void TIL_DeInitialize(void)
```

Parameters

None

Description

This function de-initializes TIL (that is, sets TIL pointer to NULL).

Return Value

None

3.1.3 TIL_StartTransport()

Initializes the Shim Layer.

Synopsis

```
PSHIM TIL_StartTransport(
    PTIL pTil,
    eShimID ShimID,
    APP_CALLBACK pAppCallback)
```

Parameters

pTil [in] — Pointer to TIL

ShimID [in] — Shim Id

pAppCallback [in] — Application callback function

Description

This function initializes the Shim identified by the Shim Id and registers the application callback function.

Return Value

- Shim pointer if success
- NULL if unsuccessful

Note:

This API is used within the library, so the application should avoid using this API.

3.1.4 TIL_StopTransport()

De-Initializes the Shim Layer.

Synopsis

```
ERR_CODE TIL_StopTransport(
    PTIL pTil,
    eShimID ShimID)
```

Parameters

pTil [in] — Pointer to TIL

ShimID [in] — Shim Id

Description

This function de-initializes the Shim identified by the Shim Id.

Return Value

- **ERROR_SUCCESS** (success)
- **ERR_UNINITIALIZED_SHIM** (shim already uninitialized)

Note:

This API is used within the library, so the application should avoid using this API.

3.1.5 TIL_SendAPDU()

Sends data through the Shim.

Synopsis

```
ERR_CODE TIL_SendAPDU(
    PTIL pTil,
    eShimID ShimID,
    boolean metadata,
    uint_8 num_tfr,
    uint_8 current_qos,
    PTR_BUFFSTACK pBuffStack)
```

Parameters

pTil [in] — Pointer to TIL

ShimID [in] — Shim Id

metadata [in] — Metadata flag

num_tfr [in] — Number of transfers

current_qos [in] — Data quality of service

pBuffStack [in] — Pointer to the send buffer

Description

This function identifies the Shim by the Shim id and sends data through Shim using the input parameters.

Return Value

- **ERROR_SUCCESS** (success)
- **ERR_UNINITIALIZED_SHIM** (shim uninitialized)
- **ERR_SEND_FAILED** (failed to send data)

Note:

This API is used within the library, so the application should avoid using this API.

3.1.6 TIL_RecvApdu()

Receives data from the Shim.

Synopsis

```
ERR_CODE TIL_RecvApdu(
    PTIL pTil,
    eShimID ShimID,
    uint_8 current_qos,
    PTR_BUFFSTACK pBuffStack)
```

Parameters

pTil [in] — Pointer to TIL

ShimID [in] — Shim Id

current_qos [in] — Data quality of service

pBuffStack [in] — Pointer to the receive buffer

Description

This function identifies the Shim by the Shim Id and receives data from the Shim using the input parameters.

Return Value

- **ERROR_SUCCESS** (success)
- **ERR_UNINITIALIZED_SHIM** (shim uninitialized)
- **ERR_RECV_FAILED** (failed to receive data)

Note:

This API is used within the library, so the application should avoid using this API.

Chapter 4
Medical Connectivity Library API

This section discusses the Medical Connectivity Library API functions in detail.

4.1 Medical Connectivity Library API Function Listings

4.1.1 Ieee11073Initialize()

Initializes the Medical Connectivity Library.

Synopsis

```
ERR_CODE Ieee11073Initialize(
    PTIL pTil,
    eShimID ShimID,
    MED_APP_CALLBACK pfnAppCallback)
```

Parameters

pTil [in] — Pointer to TIL

ShimID [in] — Pointer to Shim

pfnAppCallback [in] — Application callback pointer

Description

This function initializes the Medical Connectivity Library, starts the transport identified by the Shim pointer and registers a callback function to the application.

Return Value

- **ERROR_SUCCESS** (success)
- **ERR_GENERAL** (start transport failed)

4.1.2 AgentSendAssociationRequest()

Sends association request packet to the Shim Layer.

Synopsis

```
ERR_CODE AgentSendAssociationRequest(DataProtoList* pDataProtoList)
```

Parameters

pDataProtoList [in] —Pointer to the data proto list

Description

This function adds the association request packet header to the data proto list and sends the association request packet to the Shim Layer to be transported to the manager.

Return Value

- **ERROR_SUCCESS** (success)
- **ERR_INVALID_REQUEST** (device not in a state to send association request)
- **ERR_INSUFFICIENT_MEMORY** (if unable to add header)

4.1.3 AgentSendAssociationReleaseRequest()

Sends association release request packet to the Shim Layer.

Synopsis

```
ERR_CODE AgentSendAssociationReleaseRequest(Release_request_reason RelReqRes)
```

Parameters

RelReqRes [in] — Reason for releasing association

Description

This function adds the association release request packet header to the release request reason and sends the association release request packet to the Shim Layer to be transported to the manager.

Return Value

- **ERROR_SUCCESS** (success)
- **ERR_INVALID_REQUEST** (device not in a state to send association release request)
- **ERR_INSUFFICIENT_MEMORY** (if unable to add header)

4.1.4 AgentSendMeasurements()

Sends measurements taken by the application to the manager.

Synopsis

```
ERR_CODE AgentSendMeasurements(
    ObservationScanList* (*pObsScanList)[],
    HANDLE handle,
    intu8 ReportType,
    intu16 ScanCount,
    intu8 bConfirm)
```

Parameters

pObsScanList [in] — Pointer to an array of observation scan lists

handle [in] —Handle of the object which has to send measurements

ReportType [in] — Type of report

ScanCount [in] —Number of scans

bConfirm [in] — True for confirmed event report

Description

This function validates and sends the measurements given by the observation scan list array via the object specified by the handle in the format given by the report type. This API is used to send measurements when the device does not have support for multi persons.

Return Value

- **ERROR_SUCCESS** (success)
- **ERROR_INVALID_PARAM** (input parameters incorrect)
- **ERROR_INVALID_DATA** (data in observation Scan list is incorrect)
- **ERR_INSUFFICIENT_MEMORY** (memory constraint)

4.1.5 AgentSendPersonMeasurements()

Sends multi person measurements taken by the application to the manager.

Synopsis

```
ERR_CODE AgentSendPersonMeasurements(
    ScanReportPerVarList* (*pScanRptPerVarList)[],
    HANDLE handle,
    intu8 ReportType,
    intu16 ScanCount,
    intu8 bConfirm)
```

Parameters

pScanRptPerVarList [in] — Pointer to an array of Scan report per var list

handle [in] —Handle of the object which has to send measurements

ReportType [in] — Type of report

ScanCount [in] —Number of scans

bConfirm [in] — True for confirmed event report

Description

This function validates and sends the measurements given by the scan report per var list array via the object specified by the handle in the format given by the report type. This API is used to send measurements when the device has multi person support.

Return Value

- **ERROR_SUCCESS** (success)
- **ERROR_INVALID_PARAM** (input parameters incorrect)
- **ERROR_INVALID_DATA** (data in Scan report per var list is incorrect)
- **ERR_INSUFFICIENT_MEMORY** (memory constraint)

4.1.6 AddEntryToObsScanList()

Adds an entry to the observation scan list.

Synopsis

```
ERR_CODE AddEntryToObsScanList(
    HANDLE handle,
    OID_Type AttrId,
    intu16 AttrLen,
    void* pAttrVal,
    ObservationScanList* pObsScanList)
```

Parameters

handle [in] —Handle of the object whose measurement is taken

AttrId [in] — Attribute ID

AttrLen [in] —Attribute Length

pAttrVal [in] — Pointer to the attribute value

pObsScanList [out] — Pointer to the buffer where entry should be added

Description

This function creates or adds entry to the observation scan list based on the input parameters. This API is used to create observation scan list when the device sends data using the 4.1.4 “AgentSendMeasurements()."

Return Value

- **ERROR_SUCCESS** (success)
- **ERROR_INVALID_PARAM** (input parameters incorrect)

4.1.7 AddEntryToScanRptPerVarList()

Adds an entry to the Scan report per var list.

Synopsis

```
ERR_CODE AddEntryToScanRptPerVarList (
    HANDLE handle,
    PersonId PersonID,
    OID_Type AttrId,
    intu16 AttrLen,
    void* pAttrVal,
    ScanReportPerVarList* pScanRptPerVarList)
```

Parameters

handle [in] —Handle of the object whose measurement is taken

PersonID [in] —ID of the person whose measurements is taken

AttrId [in] — Attribute ID

AttrLen [in] —Attribute Length

pAttrVal [in] — Pointer to the attribute value

pScanRptPerVarList [out] — Pointer to the buffer where entry should be added

Description

This function creates or adds entry to the Scan report per var list based on the input parameters. This API is used to create scan report per var list when the device sends data using the 4.1.5 “AgentSendPersonMeasurements()."

Return Value

- **ERROR_SUCCESS** (success)
- **ERROR_INVALID_PARAM** (input parameters incorrect)

4.1.8 UpdatePmSegmentEntry()

Updates Pm Segment by adding an entry to the Pm Segment.

Synopsis

```
ERR_CODE UpdatePmSegmentEntry(
    HANDLE Handle,
    InstNumber InstNum,
    ObservationScanList* pObsScanList)
```

Parameters

handle [in] — Handle to the PM Store

InstNum [in] — PM Segment Instance Number

pObsScanList [in] — Pointer to the Observation Scan List

Description

This function adds entry to the PM Segment identified by the PM store handle and PM Segment instance number.

Return Value

- **ERROR_SUCCESS** (success)
- **ERR_INSUFFICIENT_MEMORY** (memory constraint)
- **ERR_INVALID_REQUEST** (operational state of pm segment is disabled)
- **ERROR_INVALID_DATA** (data in observation scan list is incorrect)
- **ERROR_INVALID_PARAM** (input parameters incorrect)

4.1.9 UpdatePmSegmentEntry()

Sends Pm Segment data to the manager.

Synopsis

```
ERR_CODE SendSegmentData(
    SegmentDataEvent* pSegmDataEvent,
    HANDLE PmStoreHandle)
```

Parameters

pSegmDataEvent [in] — Pointer to the PM Segment data

PmStoreHandle [in] — Handle to the PM Store

Description

This function sends segment data to the Shim Layer to be transported to the manager.

Return Value

- **ERROR_SUCCESS** (success)
- **ERR_INSUFFICIENT_MEMORY** (memory constraint)

Chapter 5
Data Structures

This section discusses the data structures that are passed as parameters in the various API functions.

5.1 Data Structure Listings

5.1.1 MED_APP_CALLBACK()

This callback function is called for generic events and is passed as an input parameter to 4.1.1 “Ieee11073Initialize()” from the application to the Medical Connectivity Library. The *event_id* input parameter states the type of event. The *pvoid* parameter passed to the function contains information about the event. The information passed through the *pvoid* parameter is based on the type of the event. The application implementing this callback typecasts the data parameter to the data type or structure based on the type of the event before reading it.

Synopsis

```
typedef void(_CODE_PTR_ MED_APP_CALLBACK)(
    IEEE11073_EVENT event_id,
    void* pvoid);
```

Callback Parameters

event_id — Type of event

pvoid — Event data based on the *event_id* value

5.1.2 PPMSEGDATAXFER

This structure defines information about the PM Segment instance number and information about the PM Store to which it belongs.

Synopsis

```
typedef struct _PMSEGDATAXFER
(
        HANDLE Handle;
        InstNumber SegInstNum;
) PMSEGDATAXFER, *PPMSEGDATAXFER;
```

Fields

Handle — Handle of the PM Store Object

SegInstNum — Instance number of the PM Segment

5.1.3 PTRIGSEGMDATAXFRRSP

This structure defines information about PM Segment identification and is passed by the Medical Connectivity Library to the application as a parameter of the application callback when PM Segment data transfer is triggered by the manager. The application should set the trigger response depending on whether segment has data or is empty.

Synopsis

```
typedef struct _TRIGSEGMDATAXFRRSP
(
        HANDLE Handle;
        TrigSegmDataXferRsp TrigSegmDataRsp;
) TRIGSEGMDATAXFRRSP, *PTRIGSEGMDATAXFRRSP;
```

Fields

Handle — Handle of the PM Store object

TrigSegmDataRsp — Triggered segment data response

5.1.4 PCLRPMSEGMINFO

This structure defines information about the PM Segment instance number and information about the PM Store to which it belongs.

Synopsis

```
typedef struct _CLRPMSEGMINFO
(
        HANDLE Handle;
        InstNumber SegInstNum;
) CLRPMSEGMINFO, *PCLRPMSEGMINFO;
```

Fields

Handle — Handle of the PM Store object

SegInstNum — Instance number of the PM Segment

5.1.5 IEEE11073_EVENT

This event is passed as an input parameter to the application callback.

Synopsis

```
typedef  enum _IEEE11073_EVENT
{
        IEEE11073_TRANSPORT_CONNECT,
        IEEE11073_TRANSPORT_DISCONNECT,
        IEEE11073_ASSOCIATION_RELEASING,
        IEEE11073_ASSOCIATION_RELEASED,
        IEEE11073_CONFIGURATION_TIMEDOUT,
        IEEE11073_CONFIG_REJECTED,
        IEEE11073_ERROR,
        IEEE11073_REJECT,
        IEEE11073_ABORT,
        IEEE11073_OPERATING,
        IEEE11073_EVNTRPT_SENT,
        IEEE11073_PERIODIC_SCANNER_EVENT,
        IEEE11073_CLEAR_PMSEGMENT,
        IEEE11073_TRIG_PMSEGMENT,
        IEEE11073_INTIALIZE_DIM,
        IEEE11073_GET_DATAPROTO,
        IEEE11073_INITIALIZE_DIM_FAILED,
        IEEE11073_EVENTRPT_TIMEDOUT
} IEEE11073_EVENT;
```

Enum Values

Enum Value	Description
IEEE11073_TRANSPORT_CONNECT	When the device is successfully connected to any transport, there is an application callback with event id as IEEE11073_TRANSPORT_CONNECT.
IEEE11073_TRANSPORT_DISCONNECT	When the device is disconnected from the transport, the application gets a callback with event id as IEEE11073_TRANSPORT_DISCONNECT.
IEEE11073_ASSOCIATION_RELEASING	This event is received by the application when the device has sent or received an association release request.
IEEE11073_ASSOCIATION_RELEASED	This event is received by the application when the association between the device and the manager is released. On receiving this event, the application can establish the association again by sending an association request to the manager.
IEEE11073_CONFIGURATION_TIMEDOUT	This event is received by the application when the device did not receive the response to the configuration event report within the timeout specified by the IEEE11073-20601 specifications. The device is no longer associated with the manager and the application can establish the association again by sending an association request to the manager.
IEEE11073_CONFIG_REJECTED	This event is received by the application when a device configuration is rejected by the manager.
IEEE11073_ERROR	This event is received by the application whenever an error result packet is received by the device.

Enum Value	Description
IEEE11073_REJECT	This event is received by the application whenever a reject result packet is received by the device.
IEEE11073_ABORT	This event is received by the application whenever an abort packet is sent or received by the device. The device is no longer associated with the manager and the application can establish the association again by sending an association request to the manager.
IEEE11073_OPERATING	This event is received by the application when the device has successfully established an association with the manager and a device configuration is accepted by the manager. The application can start sending measurements after the device has reached operating state.
IEEE11073_EVNTRPT_SENT	This event is received by the application whenever any event report is sent over the transport.
IEEE11073_PERIODIC_SCANNER_EVENT	This event is received by the application periodically with the period equal to the reporting interval of the periodic scanner, if any.
IEEE11073_CLEAR_PMSEGMENT	This event is received by the application whenever any request to clear a PM Segment is received by the device. The application should clear the contents of the required PM Segment.
IEEE11073_TRIG_PMSEGMENT	This event is received by the application whenever any request to send any PM Segment data is received by the device. The application should return whether the PM segment has data or is empty. If it has data, the application should send the segment data.
IEEE11073_INTIALIZE_DIM	The application should return pointer to the DIM upon receiving this event.
IEEE11073_GET_DATAPROTO	This event is received by the application when the Medical Connectivity Library needs a pointer to the data proto list. The application should return the required pointer.
IEEE11073_INTIALIZE_DIM_FAILED	This event is received by the application whenever DIM initialization fails.
IEEE11073_EVENTRPT_TIMEDOUT	This event is received by the application when device does not receive EVENT Report Response within the timeout specified by the IEEE11073-20601 specifications. The device is no longer associated with the manager and the application can establish the association again by sending an association request to the manager.

5.1.6 TRANSPORTEVENTID

This enumerated data type specifies different events that are listened to by the Transport Layer.

Synopsis

```
typedef enum _TRANSPORTEVENTID
{
        TRANSPORT_CONNECT = 0,
        TRANSPORT_DISCONNECT,
        TRANSPORT_DATARECIEVED,
        TRANSPORT_DATASENDCOMPLETE,
        TRANSPORT_GETDATABUFFER,
        TRANSPORT_GET_XFER_SIZE
}TRANSPORTEVENTID;
```

Enum Values

Enum Value	Description
TRANSPORT_CONNECT	This event is received by the Transport Layer whenever a connection is established with any transport.
TRANSPORT_DISCONNECT	This event is received by the Transport Layer whenever a connection with any transport is disconnected.
TRANSPORT_DATARECIEVED	This event is received by the Transport Layer whenever a complete APDU is received by the Shim.
TRANSPORT_DATASENDCOMPLETE	This event is received by the Transport Layer whenever a packet send is completed over a transport.
TRANSPORT_GETDATABUFFER	This event is received by the Transport Layerr to assign and pass a buffer for a given size of data.
TRANSPORT_GET_XFER_SIZE	This event is received by the Transport Layer to calculate the total size of the APDU.

5.1.7 PTR_TIL_XFER_SIZE

This structure is passed by Shim Agent to TIL to request for Transfer Size.

Synopsis

```
typedef struct _TIL_XFER_SIZE
{
        uint_8_ptr in_buff;
        uint_16 in_size;
        uint_16 transfer_size;
}TIL_XFER_SIZE, *PTR_TIL_XFER_SIZE;
```

Fields

in_buff — Pointer to buffer

in_size — Length of the buffer

transfer_size — Transfer size

5.1.8 PTR_TIL_RX_BUFF

Transport Independent Layer receive buffer structure.

Synopsis

```
typedef struct _TIL_RX_BUFF
{
        uint_16 in_size;
        uint_8_ptr in_buff;
        uint_16 out_size;
        uint_8_ptr out_buff;
        boolean meta_data_packet;
}TIL_RX_BUFF, *PTR_TIL_RX_BUFF;
```

Fields

in_size — Size of input buffer

in_buff — Pointer to input buffer

out_size — Size of output buffer

out_buff — Pointer to output buffer

meta_data_packet — Meta data packet flag

5.1.9 APP_CALLBACK()

Application callback function type

Synopsis

```
typedef void*(_CODE_PTR_ APP_CALLBACK)(
    TRANSPORTEVENTID event_id,
    void* pArg);
```

Fields

event_id — Events that are listened to by the transport layer

pArg — Event data based on the Event Id value

5.1.10 PFN_SHIM_INITIALIZE()

Shim initialize function type

Synopsis

```
typedef ERR_CODE (_CODE_PTR_ PFN_SHIM_INITIALIZE)(
    APP_CALLBACK pAppCallback);
```

Fields

pAppCallback — Pointer to the application callback function

5.1.11 PFN_SHIM_DEINITIALIZE()

Shim de-initialize function type

Synopsis

```
typedef ERR_CODE (_CODE_PTR_ PFN_SHIM_DEINITIALIZE)(void);
```

Fields

None

5.1.12 PFN_SHIM_SEND_DATA()

Shim send data function type

Synopsis

```
typedef ERR_CODE (_CODE_PTR_ PFN_SHIM_SEND_DATA)(
    boolean meta_data,
    uint_8 num_tfr,
    uint_8 current_qos,
    PTR_BUFFSTACK pBuffStack);
```

Fields

meta_data — Meta data packet flag

num_tfr — Number of transfers

current_qos — Data QoS

pBuffStack — Pointer to the send buffer stack

5.1.13 PFN_SHIM_RECV_DATA()

Shim receive function type

Synopsis

```
typedef ERR_CODE (_CODE_PTR_ PFN_SHIM_RECV_DATA)(
    uint_8 current_qos,
    PTR_BUFFSTACK pBuffStack);
```

Fields

current_qos — Data QoS

pBuffStack — Pointer to the receive buffer stack

5.1.14 eShimID

This enumerated data type specifies Shim Ids.

Synopsis

```
typedef enum
{
        SHIM_USB,
        SHIM_SERIAL,
        SHIM_TCP_ID,                    /* Currently Not Supported */
        SHIM_BLUETOOTH                  /* Currently Not Supported */
}eShimID;
```

Enum Values

Enum Value	Description
SHIM_USB	USB Shim
SHIM_SERIAL	Serial Shim
SHIM_TCP_ID	TCP Shim (currently not supported)
SHIM_BLUETOOTH	Shim bluetooth (currently not supported)

5.1.15 PSHIM

Shim interface structure

Synopsis

```
typedef struct _SHIM
{
        /* SHIM ID*/
        eShimID ShimId;
        /* Initialize Shim */
        PFN_SHIM_INITIALIZE pfnShimInitialize;
        /* Deinitialize Shim */
        PFN_SHIM_DEINITIALIZE pfnShimDeInitialize;
        /* Send Data */
        PFN_SHIM_SEND_DATA pfnShimSendData;
        /* Receive Data */
        PFN_SHIM_RECV_DATA pfnShimRecvData;
}SHIM, *PSHIM;
```

Fields

ShimId — Shim Id

pfnShimInitialize — Initialize Shim

pfnShimDeInitialize — De-initialize Shim

pfnShimSendData — Send data

pfnShimRecvData — Receive data

5.1.16 PTIL

Transport independent layer structure

Synopsis

```
typedef struct _TIL
{
        uint_8 ShimCount;
        PSHIM (*aShim)[];
}TIL, *PTIL;
```

Fields

ShimCount— Shim count

aShim — Array of Shim pointers

Chapter 6
PHDC Host Class API

This section describes the PHDC Host class interface functions.

6.1 Introduction

The PHDC purpose is to enable seamless interpretability between personal health care devices (such as glucose meters, pulse oximeters, thermometers, etc.) and USB hosts. The USB Class definition for personal health care devices provides a generic mechanism by which standardized messages can be sent over USB.

6.2 Features

The PHDC Host class driver provides an interface to the USB Host controller, allowing the application layer to handle the data exchange with the IEE 11073 Agent using standard PHDC commands in the scope of gathering the personal health care data.

The PHDC Host class provides the following functionalities:

- Manages a class interface with the connected device consisting in 3 communication pipes corresponding to the attached device endpoints (1 Bulk IN, 1 Bulk OUT endpoint and 1 Interrupt IN Endpoint)
- PHDC data sending with Metadata support
- PHDC data receiving with Metadata support
- PHDC Send Class Request function with SET_FEATURE, CLEAR_FEATURE, GET_STATUS requests support
- Send Complete Event indication to the application layer
- Receive Complete Event indication to the application layer
- Send Control Requests Complete Event indication to the application layer

6.3 PHDC Host Constants

6.3.1 PHDC specific status codes

The following PHDC specific status codes are passed to the application through the event complete indication callbacks:

```
#define USB_PHDC_RX_OK                  0x00
#define USB_PHDC_TX_OK                  0x00
```

```
#define USB_PHDC_CTRL_OK                    0x00

#define USB_PHDC_RX_ERR_METADATA_EXPECTED   0x01
#define USB_PHDC_RX_ERR_DATA_EXPECTED       0x02

#define USB_PHDC_ERR                        0x7F

#define USB_PHDC_ERR_ENDP_CLEAR_STALL       0xFF
```

In case of a successful PHDC transfer, the event indication will be called with 0x00 (RX_OK/TX_OK/CTRL_OK) as the PHDC specific status.

The `USB_PHDC_RX_ERR_METADATA_EXPECTED` and `USB_PHDC_RX_ERR_DATA_EXPECTED` indicates that the initiated Rx operation has finished with error. The host received plain data while it was expecting for metadata or the host received plain data while expecting metadata. However, the received data is fully available for the application to process if this chooses to ignore this error.

The `USB_PHDC_ERR` indicates that the USB host stack encountered an error while processing the initiated transfer. This error is also transmitted to the event complete indication using the USB standard status codes.

The `USB_PHDC_ERR_ENDP_CLEAR_STALL` indicates that the PHDC host attempted to clear the device Endpoint STALL status and failed.

6.3.2 PHDC control request types

The following definitions are used by the usb_class_phdc_send_control_request function to identify the PHDC control request:

```
#define PHDC_GET_STATUS_BREQ            0x00
#define PHDC_CLEAR_FEATURE_BREQ         0x01
#define PHDC_SET_FEATURE_BREQ           0x03
```

6.4 PHDC data types

This section describes the main C-structures and data types used by the PHDC host class.

Synopsis

```
USB_PHDC_PARAM
```

Definition

```
typedef struct usb_phdc_param_type {
  CLASS_CALL_STRUCT_PTR          ccs_ptr;
  uint_8                         classRequestType;
  boolean                        metadata;
  uint_8                         qos;
  uint_8*                        buff_ptr;
  uint_32                        buff_size;
  uint_32                        tr_index;
  _usb_pipe_handle               tr_pipe_handle;
  uint_8                         usb_status;
```

```
  uint_8                          usb_phdc_status;
} USB_PHDC_PARAM;
```

Description

PHDC required type for the parameter passing to the PHDC transfer functions (Send / Receive/ Ctrl). A pointer to this type is required when those functions are called, pointer which will be also transmitted back to the application when the corresponding callback function is called by the PHDC through the callback_param_ptr.

The application can maintain a linked list of transfer requests pointers, knowing at any moment what the pending transactions with the PHDC are.

Structure elements

- ccs_ptr: pointer to CLASS_CALL_STRUCT which identifies the interface.
- class_Request_type: The type of the PHDC request (SET_FEATURE / CLEAR_FEATURE / GET_STATUS). This parameter is used only by the usb_class_phdc_send_control_request function.
- metadata: Boolean indicating a metadata send transfer. Used only by the *usb_class_phdc_send_data* function.
- QoS: The qos for receive transfers. Used only by the *usb_class_phdc_recv_data* function.
- buffer_ptr: Pointer to the buffer used in the transfer. Used only by the send and receive functions (usb_class_phdc_send_data / usb_class_phdc_recv_data)
- buff_size: The size of the buffer used for transfer. Used only by the send and receive functions (*usb_class_phdc_send_data / usb_class_phdc_recv_data*).
- tr_index: Unique index which identifies the transfer after is queued in the USB Host API lower layers. This parameter is written by PHDC in case of a Send / Receive transfer (only if USB_STATUS is USB_OK).
- tr_pipe_handle: The handle on which the transfer was queued. This parameter is written by PHDC in case of a Send / Receive transfer (only if USB_STATUS is USB_OK).
- usb_status: standard USB_STATUS when the transfer is finished (the application callback is called). This parameter is written by the PHDC when a Send / Recv / Ctrl transfer is finished. Not valid until the corresponding callback is called.
- usb_phdc_status: the PHDC specific status code for the current transaction. Can take the following values: PHDC specific status codes. This parameter is written by the PHDC when a Send / Recv / Ctrl transfer is finished. Not valid until the corresponding callback is called.

Synopsis

```
typedef  void  (* phdc_callback)(USB_PHDC_PARAM *call_param);
```

Description

Function pointer keeping the current transaction parameters. It contains a pointer to a USB_PHDC_PARAM struct.

6.5 PHDC function listing

6.5.1 usb_class_phdc_init

Synopsis

```
void usb_class_phdc_init
   (
      /* [IN]  structure with USB pipe information on the interface */
      PIPE_BUNDLE_STRUCT_PTR      pbs_ptr,

      /* [IN] phdc call struct pointer */
      CLASS_CALL_STRUCT_PTR       ccs_ptr
   )
```

Parameters

pbs_ptr [IN] —Pointer to the pipe bundle structure containing USB pipe information for the attached device.

ccs_ptr [IN] —phdc call structure pointer. This structure contains a class validity-check code and a pointer to the current interface handle.

Description

This function serves the main purpose of initializing the PHDC interface structure with the attached device specific information containing descriptors and communication pipes handles.

The usb_class_phdc_init function is usually called by the common-class layer services as the result of an interface select function call from the Application / IEEE 11073 Manager. The application will select the interface after the USB_ATTACH indication event from the USB host API.

Return Value

None

6.5.2 usb_class_phdc_set_callbacks

Synopsis

```
USB_STATUS usb_class_phdc_set_callbacks
   (
     CLASS_CALL_STRUCT_PTR       ccs_ptr,
     phdc_callback  sendCallback,
     phdc_callback  recvCallback,
     phdc_callback  ctrlCallback
   )
```

Parameters

ccs_ptr [IN] —pointer to the current phdc interface instance for which the callbacks are set.

sendCallback [IN] —function pointer for the send Callback function.

recvCallback [IN] —function pointer for the receive Callback function.

ctrlCallback [IN] —function pointer for the send Control Callback function.

Description

The usb_class_phdc_set_callbacks function is used to register the application defined callback functions for the PHDC send, receive and control request actions. Providing a non-NULL pointer to a callback function (phdc_callback type) will register the provided function to be called when the corresponding action is complete, while providing a NULL pointer will invalidate the callback for the corresponding action.

The applications registered callbacks are unique for each selected PHDC interface. Only one Send callback and one Receive callback can be registered for each PHDC interface. Because the PHDC class supports multiple send / receive actions to be queued in the lower layers at the same time, the application can identify the action for which the callback function was called by using the call_param pointer which can point to a different location for each Send/Receive/Ctrl function call. The call_param pointer is transmitted as parameter to the PHDC Send/Receive/Ctrl functions and given back to the application when the Send/Receive/Ctrl callback function is called.

Before saving the callback pointers in the PHDC interface structure, the *usb_class_phdc_set_callbacks* function verifies all the transfer pipes for pending transactions. The callbacks for send / receive actions cannot be changed while there are pending transactions on the pipes. In this case, the function will deny the set callbacks request and will return USBERR_TRANSFER_IN_PROGRESS.

If the pipes have no pending transactions, the usb_class_phdc_set_callbacks function will save the callbacks pointers in the current interface structure and will return USB_OK.

At USB transfer completion, the user registered callbacks (sendCallback, recvCallback or controlCallback) will be called from the PHDC class after the internal processing of the transfer status and using the provided callback_param at the action start.

Return Value

- USB_OK (success)
- USBERR_NO_INTERFACE (the provided interface is not valid)
- USBERR_TRANSFER_IN_PROGRESS (as there are still pending transfers on the data pipes, the request to register the callbacks was denied. No previously registered callback was affected)

6.5.3 usb_class_phdc_send_control_request

Synopsis

```
USB_STATUS usb_class_phdc_send_control_request
   (
      USB_PHDC_PARAM *call_param
   )
```

Parameters

call_param [IN]: pointer to a USB_PHDC_PARAM structure.

Description

The *usb_class_phdc_send_control_request* function is used to send PHDC class specific request to the attached device. As defined by the PHDC class specification, the request must be one of the following types: SET_FEATURE, CLEAR_FEATURE, GET_STATUS (on page 6-1).

SET_FEATURE, CLEAR_FEATURE requests:

In order not to stall the device endpoint, the usb_class_phdc_send_control_request function will first verify if the attached device supports Meta Data preamble transfer feature for the SET_FEATURE and CLEAR_FEATURE request. If the preamble capability is not supported, this function will return USBERR_INVALID_REQ_TYPE and exit.

Only one SET_FEATURE/CLEAR_FEATURE control requests to the device can be queued on the control pipe at the time. In case there is another request pending, this function will deny the request by returning USBERR_TRANSFER_IN_PROGRESS. Also for the SET_FEATURE and CLEAR_FEATURE requests, this function will verify the pending transfers on the data pipes. To avoid synchronization issues with preamble, the phdc will not transmit the control request if the data pipes have transfers queued for the device. In this case, the function will return USBERR_TRANSFER_IN_PROGRESS and exit. The application is also responsible for checking the device endpoint (by issuing a GET_STATUS request) before sending a SET_FEATURE or CLEAR_FEATURE to the device.

GET_STATUS requests:

For this request, there are no restrictions in terms of pending requests on the control pipe as the GET_STATUS request will not interfere with the other PHDC send/receive function nor will cause sync issues on the device.

PHDC Send Control Callback:

The completion of the PHDC control request is managed internally by the PHDC class for handling also the device endpoint stall situation. If the PHDC is informed by the USB Host API that the device control endpoint is stalled, then the PHDC will attempt to clear the endpoint STALL by issuing a standard CLEAR_FEATURE command request to the device.

In the end, the PHDC calls the application registered callback for the control request function, using the USB provided status code, and the PHDC class status code (through the `call_param->usb_status` pointer).

If the PHDC fails to clear the endpoint stall it will call the application send control callback with the PHDC status of USB_PHDC_ERR_ENDP_CLEAR_STALL.

Return Value

- USB_OK / USB_STATUS_TRANSFER_QUEUED (success)

- USBERR_NO_INTERFACE (the provided interface is not valid)
- USBERR_ERROR (parameter error)
- USBERR_INVALID_REQ_TYPE (invalid type for the request)
- USBERR_TRANSFER_IN_PROGRESS (a control request SET / CLEAR_FEATURE is already in progress)

6.5.4 usb_class_phdc_recv_data

Synopsis

```
USB_STATUS usb_class_phdc_recv_data
   (
      USB_PHDC_PARAM *call_param
   )
```

Parameters

call_param [IN]: pointer to a USB_PHDC_PARAM structure.

Description

The *usb_class_phdc_recv_data* function is used for receiving PHDC class specific data or metadata packets. It schedules an USB receive on the QoS —selected pipe for the lower Host API. The receive transfer will end when the host has received the specified amount of bytes or if the last packet received is less than `pipe maximum packet size (MAX_PACKET_SIZE) indicating that the device doesn't have more data to send`.

Before scheduling the receive action, this function will first validate the provided call_param pointer and Rx relevant fields, by checking the *`call_param->ccs_ptr`* (class interface), *`call_param->qos`* (QoS bitmap used to identify the pipe for receive), the *`call_param->buff_ptr`* (buffer for storing the data received —cannot be NULL) and *`call_param->buff_size`* (number of bytes to receive —cannot be 0). If all the parameters are valid, the function checks if a SET_FEATURE or CLEAR_FEATURE control request is pending. If it is, the function returns USBERR_TRANSFER_IN_PROGRESS and the transaction is refused (the PHDC does not know if the device has metadata feature enabled or not in order to decode the received packet).

NOTE:

In order to prevent memory alignment issues on certain platforms, it is recommended that the provided receive size (call_param->buff_size) to be always multiple of 4 bytes.

If all the checks are passing, this function initiates an USB Host receive action on the designated pipe and registers a PHDC internal callback to handle the finishing of the Tx action.

PHDC Receive Callback:

The PHDC internal Receive Callback will be called when the USB Host API reception completes. The callback will parse the received data, populate the PHDC status codes in the USB_PHDC_PARAM

structure and call the user defined receive callback (the function registered by the user using the *usb_class_phdc_set_callbacks*).

The parameters passed to the user registered callback are:

- USB_PHDC_PARAM structure.
  - Through the usb_phdc_status, this structure will inform the user if data received are metadata preamble or regular data and if metadata preamble or regular data were expected.
  - Through the usb_status, this informs the user callback about the status of the USB transfer.

The PHDC receive callback also checks the type of data received (Plain Data or Metadata) and compares it with the type of data that was expected. In case if the Host was expecting for a metadata but only plain data was received, then, according to the health care standard, the Host will issue a SET_FEATURE (ENDPOINT_HALT) followed by a CLEAR_FEATURE (ENDPOINT_HALT) on the receiving pipe.

Return Value

- USB_OK / USB_STATUS_TRANSFER_QUEUED (success)
- USBERR_NO_INTERFACE (the provided interface is not valid)
- USBERR_ERROR (parameter error)
- USBERR_TRANSFER_IN_PROGRESS (a control request SET / CLEAR_FEATURE is in progress)

6.5.5 usb_class_phdc_send_data

Synopsis

```
USB_STATUS usb_class_phdc_send_data
   (
      USB_PHDC_PARAM *call_param
   )
```

Parameters

call_param [IN]: pointer to a USB_PHDC_PARAM structure.

Description

The *usb_class_phdc_send_data* function is used for sending PHDC class specific data or metadata packets. It schedules an USB send transfer on the Bulk-Out pipe for the lower Host API.

Before scheduling the send action, this function will first validate the provided call_param pointer and Tx relevant fields, by checking the `call_param->ccs_ptr` (class interface), the `call_param->buff_ptr` (buffer for taking the data to be sent—cannot be NULL) and `call_param->buff_size` (number of bytes to send —cannot be 0). If the parameters are valid, this function validates the data buffer provided by the application for transmission.

The *usb_class_phdc_send* function expects that application provides the data buffer constructed accordingly with the metadata preamble feature. The application is responsible of forming the data packet to be sent including the metadata preamble (USB_PHDC_METADATA_PREAMBLE), if this is used.

In case if metadata is included in the packet (*call_param_ptr->metadata* is TRUE), the attached device supports metadata and the metadata feature was already set on the device using the *usb_class_phdc_send_control_request* function, then this function will validate the QoS in the transmit packet by checking its bitmap fields and also using the QoS descriptor for the PHDC Bulk-Out pipe. If the requested QoS is not supported in the descriptor, this function denies the transfer and returns USBERR_ERROR.

Before actually sending the data, this function also checks if there are pending SET / CLEAR_FEATURE requests types to the device. Until those are completed, the send function does no know if the device has the metadata preamble feature activated, so it will deny the requested transfer and return USBERR_TRANSFER_IN_PROGRESS.

If all the checks are passing, this function initiates an USB Host send action on the Bulk-Out pipe and registers a PHDC internal callback to handle the finishing of the Tx action.

PHDC Send Callback:

The PHDC internal Send Callback will be called when the USB Host API send transfer completes. The callback will populate the PHDC status codes in the USB_PHDC_PARAM structure and call the user defined receive callback (the function registered by the user using the *usb_class_phdc_set_callbacks*).

The parameters passed to the user registered callback are:

- USB_PHDC_PARAM structure.
  - the usb_phdc_status is set to USB_PHDC_TX_OK when the received status code from USB host API is USB_OK, or USB_PHDC_ERR otherwise
  - through the usb_status, this structure pointer informs the user callback about the status of the USB transfer

The device endpoint stall situation is handled also by the internal send callback. If the PHDC is informed by the USB Host API that the device endpoint is stalled, then the PHDC will attempt to clear the endpoint STALL by issuing a standard CLEAR_FEATURE command request to the device. If the PHDC fails to clear the endpoint stall it will call the application send control callback with the PHDC status of USB_PHDC_ERR_ENDP_CLEAR_STALL.

Return Value

- USB_OK / USB_STATUS_TRANSFER_QUEUED (success)
- USBERR_NO_INTERFACE (the provided interface is not valid)
- USBERR_INVALID_BMREQ_TYPE (invalid qos bitmap fields in the sending packet)
- USBERR_ERROR (parameter error / metadata checking error)
- USBERR_TRANSFER_IN_PROGRESS (a control request SET / CLEAR_FEATURE is in progress)