TSM 1.0 – User Manual - Excerpt

TERRANOVA Smart Metering Libraries

Document revision history:

Major	Minor	Date (yymmdd)	SW Version	Note
1	1	130510	0.1	First revision
1	2	130617	0.2	Structural update
1	3	130725	0.3	Architecture revision
1	4	130905	0.4	Architecture revision
1	6	131108	0.5	API reference start, object access control functions
1	8	140210	0.10	Timer, Mutex and API update
1	9	140212	0.10	Compact pdu (11291-11-3)
1	10	140408	0.11	Stack modules implementation and linking
1	11	140515	0.13	Object Views and Example code revision
1	12	140526	0.13	Object Access revision and Selective Access description
1	13	140617	0.15	Ciphering Suite description
1	14	140711	0.15	Library system
1	15	140926	1.0	Structural update
1	16	140929	1.1	Detailed association process
1	17	141114	1.2	Structural update
1	18	141218	1.2	Mutex chapter update, MISRA C 2012 matrix
1	19	150116	1.3	Generic data types map, buffer and allocator review

Table of contents

Figures

1 INTRODUCTION

The Terranova Smart Metering Libraries provides all the necessary tools to implement a smart metering system application by using the most used standard protocols in the metering context.

The core applications:

- Smart Metering Devices;
- Data Collection Systems (SAC/DCS)
- Data concentrators (Gateways)

The library is **self-contained**[1] to abstract from the surrounding environment and it is developed from scratch by using the simplest **ANSI C** form (**C89**[2]) to achieve the maximum portability across the various machine architectures and operating systems.

The library was created paying special attention with regard to the constraints imposed by the most common embedded systems – that is safety, reliability and memory footprint reduction - and also by following the **MISRA-C**[3] rules set.

[1] The library do not use external libraries or OS routines.
[2] The oldest ANSI C standard. Not every compiler fully supports the C99 standard, so C89 is the way to write portable code.
[3] MISRA-C (Motor Industry Software Association) are a set of syntactic rules developed to facilitate C code safety, portability and reliability in the context of embedded systems.

2 FEATURES

- **Self-contained** and built from scratch by using portable **ANSI C (89)** language and following the **MISRA-C** (Motor Industry Software Association) syntactic rules set.
- **Communication protocols**: DLMS, SML, HDLC, CTR.
- **Independent from the surrounding environment (ie. firmware/OS).**
- **Easy to use** and **clean** API.
- **Object Model**: an object oriented model to organize the underlying raw data. Contains a collection of the most common interface classes (30+) and allows the definition of custom ones.
- **Scalable** and **modular** protocols stack architecture.
- **Small memory** requirements.
- Optimized built in **dynamic memory allocator** (if your system do not provides one).
- Includes the most common and useful object identifiers (**OBIS**) definitions.
- Implementation of all the **standard COSEM interface classes**.
- **Ciphering suite**: AES-128/192/256, GCM, CBC, CTR.

2.1 DLMS features

- **Supported services:** Connect, Disconnect, Get, Set, Execute, Push.
- **Transparent data block transfer** for Get service.
- **GCM-AES128** message ciphering and/or message authentication.
- **Low Level** and **High Level security** connection procedure support.
- Optimization of payload data (UNI/TS-11291-11-2).
- Support for future protocol versions.

2.2 SML features

- Supported services: Connect, Disconnect, Get, Set, Exec.
- Support for future protocol versions.

3 SMART METERING

3.1 Smart meter

A smart meter is an entity that acts as a **server (slave)** application exposing to the external world some data as a collection of objects. Each object represents some kind of real (e.g. temperature) or abstract (e.g. image transfer) information related to the specific device purpose.

Representing the information as an object allows to group together related data and allows the client to have a very neat vision of what kind of data stores and what are its capabilities.

3.2 Data collection system

A data collection system (**DCS** or **SAC**[4]) is an entity that acts as a **client (master)** in a communication. The DCS interacts with a server objects to read/write attributes and to invoke remote methods. Usually the read data (readout) will be stored in a kind database for further processing.

3.3 Data concentrator

A data concentrator is an double faced entity that acts as a client on one side and as a server on the other.

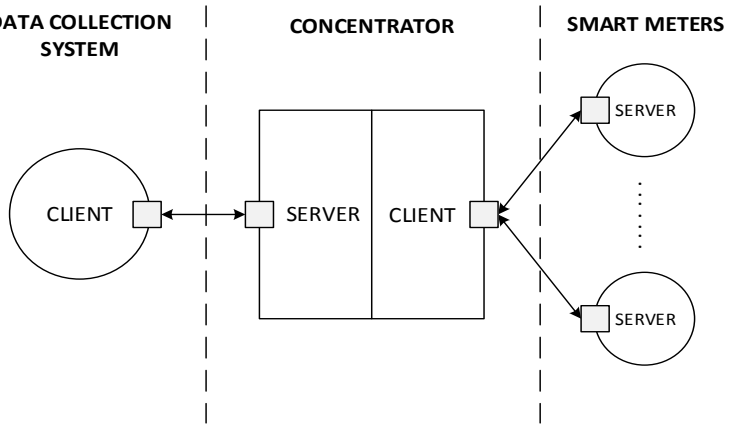


Figure 1 – client, server and concentrator

[4] Italian acronym for Central Acquisition System (Sistema di Acquisizione Centrale)

The purposes of a concentrator ranges from merely acting as a bridge between different protocols (translator) to the more sophisticated ones that implements a sort of middle level data collection system.

3.4 Peer to peer system

A peer to peer system is an hybrid system that can act both as a client and as a server by using the same communication channel. So here the differentiation between who reads and who it has been read do not exists. This scenario is pretty unusual in the smart metering context, anyway the library allows to implement such architecture.

4 GETTING STARTED

To start using the library you must understand the three core functions.

Initialization

This function takes as arguments a pointer to the activation key (size 16), the activation mode, an optional callback to asynchronously inform the user about the activation result and an optional user context to be passed to the callback.

```
typedef void (* tsm_init_cb_f)(void *ctx, tsm_res_t res);

void tsm_init(const void *key, int mode, tsm_init_cb_f cb_fun, void *cb_ctx);
```

The activation mode must be one of the following values:

```
/** Big market server (industrial). */
#define TSM_MODE_BM_SERVER   'B'
/** Mass market server (domestic). */
#define TSM_MODE_MM_SERVER   'M'
/** Data collector system client. */
#define TSM_MODE_DCS_CLIENT  'D'
/** Hand-held terminal client. */
#define TSM_MODE_HHT_CLIENT  'H'
/** Gateway or concentrator system. */
#define TSM_MODE_GATEWAY     'G'
```

Release function

Function to release the resources owned by the library and discards all the pending events in the timer queue.

```
void tsm_rlse(void);
```

Update function

This function updates the modules contexts status, check the global variables consistence, updates the timers queue and eventually triggers one or more timed events. The functions takes the number of milliseconds elapsed since the last update so it is important to a value that is more accurate as possible. If a fake or a too much approximated value is passed then the library status may become inconsistent and the results may not be the ones that the user expects.

```
void tsm_update(tsm_uint32_t ms_delta);
```

Warning

If the `malloc` implementation of your choice is the `tsm_malloc` then it is **very important** to initialize the memory allocator subsystem **before** issuing any other library call (that is true for the `tsm_init` as well).

Some utility functions are provided. These functions do not need that the library subsystem initialization.

Device identification

Function to get the library user unique identification number - 16 octets.

```
void tsm_uid(void *uid);
```

Library version

Function to get the library version number.
Even if this information is present in a header file, may happen that the user updates the library archive file without update the include tree. This is a secure way to get the correct library version number.

```
int tsm_version(void);
```

Initialization wrapper

A wrapper function is provided to help the user to initialize the library in a synchronous way.

```
tsm_res_t tsm_init2(const void *key, int mode);
```

This function is similar to the main one except that it is synchronous and the result is directly returned to the user. Pay special attention to the fact that this is a blocking one and only works if, in the meantime, there is another execution thread that calls the `tsm_update`.

Internally this is implemented by using the traditional `tsm_init`. The calling thread is blocked, by using a mutex, until the unlock callback is called by the timer. The callback also pass the result to the blocked thread.

Since this function can be very dangerous the implementation is provided to avoid confusion and deadlocks.

```
struct tsm_init2_ctx
{
    tsm_mutex_t  mutex;
    tsm_res_t    result;
};

static void cb(struct tsm_init2_ctx *ctx, tsm_res_t res)
{
```

```
    ctx->result = res;
    tsm_mutex_unlock(&ctx->mutex);
}

tsm_res_t tsm_init2(const void *key, int mode)
{
    struct tsm_init2_ctx ctx;
    ctx.mutex = TSM_MUTEX_LOCKED;
    ctx.result = TSM_RES_OTHER_ERROR;
    tsm_init(key, mode, (tsm_init_cb_f) cb, &ctx);
    tsm_mutex_lock(&ctx.mutex);      /* wait for the asynchronous unlock event */
    return ctx.result;
}
```

5 SIMPLE DATA TYPES

This section provides an overview of the core data types defined by the library.

5.1 Primitive types

A primitive data type is a type that can be directly translated (is a *typedef*) in a data type supported by the programming language. Fixed width integers are defined with the help of stdint.h standard header file.

tsm_uint8_t : one octet unsigned integer.
tsm_uint16_t : two octets unsigned integer.
tsm_uint32_t : four octets unsigned integer.
tsm_uint64_t : one octet signed integer.
tsm_int8_t : one octet signed integer.
tsm_int16_t : two octets signed integer.
tsm_int32_t : four octets signed integer.
tsm_int64_t : four octets signed integer.
tsm_float32_t : four octets floating point number (IEEE 754).
tsm_float64_t : eight octets floating point number (IEEE 754).
tsm_bool_t : one octet boolean value.
tsm_enum_t : one octet enumeration value.

5.2 Strings

Variable length sequence of octets or bits.

```
struct tsm_str
{
    tsm_uint8_t       *raw;
    tsm_size_t        size;
};
```

- **raw** - pointer to the string raw octets array.
- **size** - string length. If the string is an octet string then it represents the number of octets else if it is a bit string it represents the number of bits.

Bit string note.

In every machine architecture an octet is a collection of 8, **big endian, ordered bits** (the rightmost bit is the least significant). But in a bit string the bits are ordered in little endian.

Example. If the first two octets referenced by `raw` are $0x9D_{16}$ and $0x35_{16}$ and the `size` is 14.

first octet: $9D_{16} = 10011101_{2\ (BE)} = 10111001_{2\ (LE)}$

second octet: $B4_{16} = 00110101_{2\ (BE)} = 10101100_{2\ (BE)}$

Since the size is 14 the raw data must be interpreted as the bit string: `10111001101011`.

5.3 Date and time

Date structure

```
struct tsm_date
{
    tsm_uint8_t       year;
    tsm_uint8_t       mon;
    tsm_uint8_t       mday;
    tsm_uint8_t       wday;
};
```

- **year** - number of years since 2000. From 0 to 254 or TSM_WILDCARD_VALUE.
- **mon -** month of the year. From 1 to 12 or TSM_WILDCARD_VALUE.
- **mday -** month day. From 1 to 31 or TSM_WILDCARD_VALUE.
- **wday -** day of the week. From 1 (Monday) to 7 or TSM_WILDCARD_VALUE.

Time structure

```
struct tsm_time
{
    tsm_uint8_t       hour;
    tsm_uint8_t       min;
    tsm_uint8_t       sec;
};
```

- **hour -** hour of the day. From 0 to 23 or TSM_WILDCARD_VALUE.
- **min -** minute of the hour. From 0 to 59 or TSM_WILDCARD_VALUE.
- **sec -** seconds of the minute. From 0 to 59 or TSM_WILDCARD_VALUE.

Datetime structure

```
struct tsm_datetime
{
    tsm_uint8_t       year;
    tsm_uint8_t       mon;
    tsm_uint8_t       mday;
    tsm_uint8_t       wday;
    tsm_uint8_t       hour;
    tsm_uint8_t       min;
    tsm_uint8_t       sec;
    signed int        utc_dev: 7;
    unsigned int      is_dst  : 1;
```

```
};
```

- **utc_dev -** deviation from UTC in hours (+/- 12) or TSM_DEVIATION_NS.
- **is_dst** - boolean value to indicate if the daylight saving time is active.
- all the other fields meaning is equal to date and time

5.4 Generic data sequence

Sequence of generic data containers (ref. 4.5). The types of the single elements may be different.

```
struct tsm_array
{
    struct tsm_data   *data;
    tsm_size_t        size;
};
```

Example. We want to create a generic data structure for the following set of raw data.

```
tsm_int8_t     m1;
tsm_uint8_t    m2;
struct tsm_str m3;
```

First we define the array of `tsm_data`.

```
const struct tsm_data struct_data[] =
{
    TSM_DATA_MK(TSM_DATA_INT8, &m1),
    TSM_DATA_MK(TSM_DATA_ENUM, &m2),
    TSM_DATA_MK(TSM_DATA_OSTR, &m3)
};
```

The TSM_DATA_MK is a helper macro used to define an instance of generic data that uses indirect addressing.

Now we can create a generic data structure by using the dedicated macro or using the initialization function. If the used data array is not persistent (e.g. defined on the stack) then you must use the initialization function since it will deep copy the array content.

```
/* if the data array is persistent (eg. global, malloc, static local) */
```

```
struct tsm_array st1 = TSM_ARRAY_MK(struct_data, 3);

/* if the data array is not persistent (eg. local variable) */
struct tsm_array st2;
tsm_array_init(&st2, struct_data, 3);
```

5.5 Generic data container

Mainly used where we need a generic container for a data value. The structure have as the first member a type value and as the second member a boolean that indicates if the value is contained within the data structure itself.

```
struct tsm_data
{
    tsm_uint8_t        type;
    tsm_bool_t         direct;
    union
    {
       void            *ptr;
       tsm_uint8_t     uint8;
       tsm_uint16_t    uint16;
       tsm_uint32_t    uint32;
       tsm_int8_t      int8;
       tsm_int16_t     int16;
       tsm_int32_t     int32;
       tsm_float32_t   float32;
       struct tsm_time time;
       struct tsm_date date;
       tsm_bool_t      ubool;
       tsm_enum_t      uenum;
       tsm_uint8_t     raw[sizeof(void *)];
    } u;
};
```

- **type**: contained data type.
- **direct**: indicates if the contained value is within the data structure itself.
- **u**: when direct is TRUE the union is used to directly store values with a maximum size equal to the size of a pointer. When direct is FALSE the value can be accessed by using the `ptr` member pointer.

Like the other library structures, it can be initialized by using the initialization function, which deep copies the arguments, or created by using the helper macro.

The following table associates a type identifier in the generic data container with the type meaning and the proper underlying type to use.

Type	Type name	Proper type	Size
Null	TSM_DATA_NULL	-	0
Boolean	TSM_DATA_BOOL	tsm_bool_t	1
Enumeration	TSM_DATA_ENUM	tsm_enum_t	1
Unsigned integer	TSM_DATA_UINT8	tsm_uint8_t	1
Unsigned integer	TSM_DATA_UINT16	tsm_uint16_t	2
Unsigned integer	TSM_DATA_UINT32	tsm_uint32_t	4
Unsigned integer	TSM_DATA_UINT64	tsm_uint64_t	8
Signed integer	TSM_DATA_INT8	tsm_int8_t	1
Signed integer	TSM_DATA_INT16	tsm_int16_t	2
Singed integer	TSM_DATA_INT32	tsm_int32_t	4
Signed integer	TSM_DATA_INT64	tsm_int64_t	8
Float 32	TSM_DATA_FLOAT32	tsm_float32_t	4
Float 64	TSM_DATA_FLOAT64	tsm_float64_t	8
Octet String	TSM_DATA_OSTR	tsm_str	Variable
Visible String	TSM_DATA_VSTR	tsm_str	Variable
Bit String	TSM_DATA_BSTR	tsm_str	Variable
Date	TSM_DATA_DATE	tsm_date	4
Time	TSM_DATA_TIME	tsm_time	3
Date and Time	TSM_DATA_DATETIME	tsm_datetime	8
Array	TSM_DATA_ARRAY	tsm_array	Variable
Structure	TSM_DATA_STRUCT	tsm_array	Variable
Compact Array	TSM_DATA_CARRAY	tsm_array	Variable

6 BUFFER

Raw buffer handling structure. Mainly used to transfer data between adjacent protocol layers but can also be used whether a variable payload length storage structure is required.

```
struct tsm_buf
{
    tsm_uint16_t      delta;
    tsm_uint8_t       *rstart;
    tsm_uint8_t       *start;
    tsm_uint8_t       *end;
    tsm_uint8_t       *rend;
};
```

- **delta -** real buffer expansion/contraction alignment delta. If zero the underlying raw buffer is never resized (reallocated). If a value greater than zero is used then it must be a power of two.
- **rstart -** real buffer start pointer.
- **start** - payload start pointer.
- **end -** payload end pointer.
- **rend** - payload real end pointer.

Constraint: `rstart ≤ start ≤ end ≤ rend`.
Promise: When dynamically allocated the buffer underlying raw buffer size is always a multiple of delta.

Initialization and release functions

```
tsm_res_t tsm_buf_init(struct tsm_buf *buf, tsm_uint16_t delta,
               const void *content_src, tsm_size_t content_size,
               tsm_size_t resv_front, tsm_size_t resv_back);

void tsm_buf_rlse(struct tsm_buf *buf);
```

Note that by using the initialization function we are implicitly using a dynamic size buffer since if the passed delta is zero then it is automatically set to TSM_BUF_DELTA_DEFAULT value that is hardcoded as 16.

Four utility functions are provided to push data in the front, push data in the back, pull data from the front and pull data from the back.

```
tsm_res_t tsm_buf_push_back(struct tsm_buf *buf, const void *src, tsm_size_t n);

tsm_res_t tsm_buf_push_front(struct tsm_buf *buf, const void *src, tsm_size_t n);

tsm_res_t tsm_buf_pull_back(struct tsm_buf *buf, void *dst, tsm_size_t n);
```

```
tsm_res_t tsm_buf_pull_front(struct tsm_buf *buf, void *dst, tsm_size_t n);
```

Push consequences

Before push the data, if the free space is not enough and the buffer delta is greater than zero the buffer may be reallocated to expand its capacity. To get space the buffer payload may be also shifted around in the underlying real buffer.

Pull consequences

After the data pull, if the free space in the front or in the tail is greater that the delta then the real buffer may be resized to shrink its capacity.

The push/pull functions always maintains the raw buffer size aligned to the delta value.

Example

The buffer is initialized via the initialization function. There is a payload, a delta > 0, a reserve front space and reserve back space.

```
struct tsm_buf buf;
tsm_buf_init(&buf, 4, "whatever", 8, 3, 6);
```

On success, after the initialization, we have a buffer content allocated with the help of the dynamic memory allocator. The underlying raw buffer size, that is the payload size plus the reserve front/back size, is aligned to the nearest upper multiple of delta. We end up with:

```
buf.delta = 4;
buf.rstart = malloc(ALIGN_TO_DELTA(8+3+6, 4);        // a dynamically allocated pointer.
buf.rend = buf.rstart + ALIGN_TO_DELTA(8+3+6, 4);    // 20
buf.start = buf.rstart + 3;
buf.end = buf.start + 8;
```

Example

The buffer is initialized directly and by using a pre allocated raw buffer. Avoid reallocations by directly storing a zero delta.

```
tsm_uint8_t raw[256];
memcpy(&raw[10], "whatever", 8);
buf.rstart = raw;
buf.rend = buf.rstart+255;
buf.start = buf.rstart+10;
buf.end = buf.start+8;
buf.delta = 0;     /* do not allow reallocations */
```

7 MEMORY ALLOCATOR

Sometimes the environment where the library is used does not provide a memory allocator or simply we find that the present allocator is inefficient for our uses. In these situations, the user can use the library memory allocator: a tested and efficient pool based dynamic memory allocator.

The allocator uses a **best-fit** algorithm to minimize the memory footprint from an internal fragmentation point of view and since the acquired expansion pools are almost (direct allocations are special cases) always of the same size the external fragmentation is virtually inexistent.

Allocator functions

```
void *tsm_malloc(tsm_size_t size);

void  tsm_free(void *ptr);

void *tsm_realloc(void *ptr, tsm_size_t size);

void *tsm_calloc(tsm_size_t nmemb, tsm_size_t size);
```

[omissis]

8 TIMER

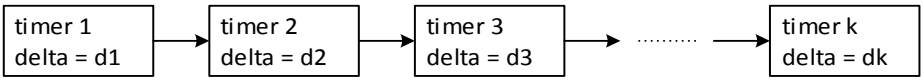
The standard way to create a timer is actually by creating a separate thread that will sleep some time at first, then wake up to check it is time for lunch; if not, sleep again otherwise call a user defined timeout notification routine.

Though the application which needs thousands of timers may sound impractical, there may be that much timers running on a more complex application. As we mentioned before, timer is eventually a separate thread. The problem is how many timers can we create on a system. Apparently, the number is not infinity. When you run 'ulimit -a' under UNIX/Linux, you will find the max number for running threads supported by this system. Then what if the number of timer are reaching the max number supported by OS? The answer is a lot of timers may fail to wake up timely.

The way to handle the case above is to using timer queue – a queue used to hold all the timers - and a thread running and checking if there is any timer expires.

The timer queue is a list of lightweight objects, known as timer-queue timers, where each one holds the time offset from the previous one. When the timer create function is called this object is created and initialized with a callback function to be called when the specified due time arrives, with a user context and with the timer type (periodic and oneshot).

Note that the timer is not immediately inserted in the queue of the running timers until the call to the start function.



The timer k time left is:

$$\text{timer}_\text{k}.\text{left} = \sum_{n=1}^{k} \text{timer}_\text{i}.\text{delta}$$

[omissis]

9 MUTEX

A mutex refers to the object used to achieve mutual exclusion, where mutual exclusion refers to the requirement of ensuring that no two concurrent processes are in the critical sections at the same time.

The critical section refers to a period when the process accesses a shared resource, such as a protocol contexts.

A mutex is a very machine dependent object and although some machine versions are implemented by the library if a version for your machine is not found you must implement the mutex functions.

Note that if you find a mutex implementation within the library this will be a **WEAK** implementation. That means that the functions are defined as weak symbols and if the user defines its own implementation then the weak ones are discarded in favor of the user definitions.

```
typedef void * tsm_mutex_t;

tsm_res_t tsm_mutex_init(tsm_mutex_t *mtx, int is_locked);

void tsm_mutex_rlse(tsm_mutex_t *mtx);

void tsm_mutex_lock(tsm_mutex_t *mtx);

void tsm_mutex_unlock(tsm_mutex_t *mtx);
```

If the same object (e.g. a protocol context or a buffer) can be handled more than one concurrent thread then, if an implementation is not provided within the library, it is very important to implement the mutex functions for your machine in the correct way. If not, the behavior of the application may become unpredictable.

[omissis]

10 CIPHERING SUITE

The library provides the following cryptographic services

- Ciphering: AES-128, AES-192, AES-256;
- Block cipher mode of operation: CBC, CTR, GCM;
- Ciphered hashing: CMAC, GMAC;
-

[omissis]

11 OBJECT MODEL

The object model provided with the library is an implementation of the object model proposed in the **IEC 62056-62 (interface classes)** normative. This normative is usually referred in the DLMS context, anyway we have found that this object model is generic and flexible enough to be used in any context where high level data representation is needed.

11.1 Abstract data type

An object class is a data structure composed by a set of **Attributes** and **Methods** created with the intent to bound together a set of related data in an object-oriented way.

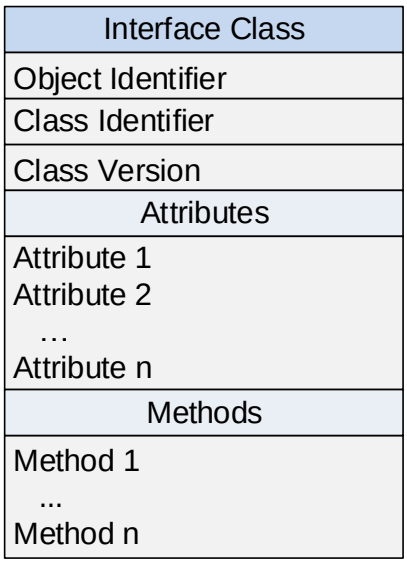


Figure 2 - abstract interface class

The standard defines plenty object types and each one is best suited than the others to represents some kind of information. For example, if you need to represent an “image transfer” process then you can use the dedicated “image transfer” object class.

Note that the first attribute of any standard object is always the object instance identifier. The standard defines this attribute as a six octets **OBIS**[5] code, used to identify what a specific object instance represents. For example, the "device identification id" and the "firmware version" objects can be both instances of the same object class but their logical names must be different.

The DLMS UA has defined a set of standard object instance identifiers to represents almost every kind of data in the metering context and also suggests which kind of object class the instance must use; anyway the user is free to define new custom objects identifiers. A partial list of the most useful standard objects identifiers, and suggestions about what kind of interface class you should use for them, it is provided within the library header files.

If you wish to use some other standard objects or to check if an OBIS is already in use by the standard you can check it by consulting the object definition table Excel file available from the DLMS official page:

http://www.dlms.com/documentation/listofstandardobiscodesandmaintenanceproces/index.html

Fortunately there is a simple free tool, called **OBIS Helper** that silently uses that Excel file and is able to find and identify the OBIS codes in a more comfortable way.

[5] OBIS: Object Identification System defines identification codes for all data in the COSEM compliant metering equipment (IEC 62056-61).

11.2 Object views

An object view is a mask of access rules through which the user can access to the object attributes and methods in a more regulated and safer way. This provides a convenient way to bound different object access policies to different association profiles. When an association changes the user just have to access the objects using the linked set of object views.

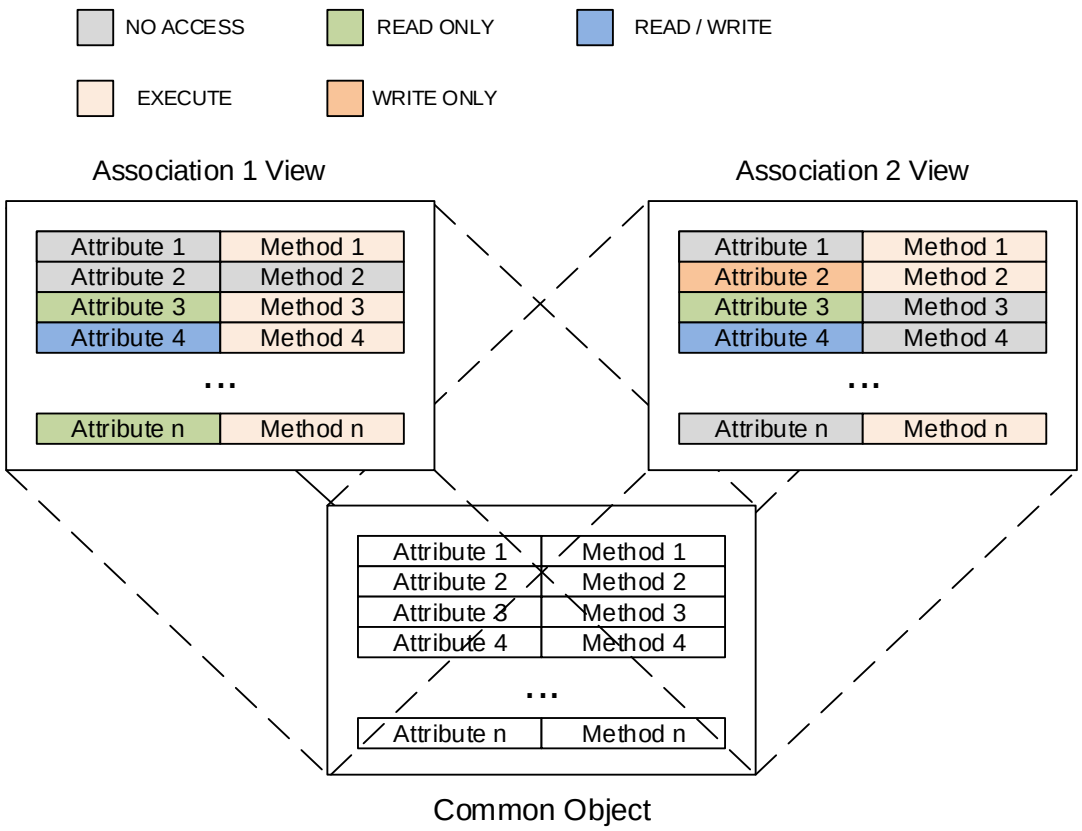


Figure 3 - object views

[omissis]

11.3 Provided interface classes

This paragraph enumerates the objects classes provided with the library. A short description of each one is included here, for a detailed description about the attributes and methods of each object class refer to the correspondent library API header file.

11.3.1 Classes for access control and management

- **Association** (tsm_ic_associaion.h): model an association profile.
- **SAP assignment** (tsm_ic_sap_assign.h): model logical structure of a physical device.
- **Security setup** (tsm_ic_security_setup.h) : contain an association security configuration.
- **Image transfer** (tsm_ic_img_trans.h): model the mechanism of transferring binary files.

11.3.2 Classes for measurement data

- **Data** (tsm_ic_data.h): generally used to store generic data.
- **Register** (tsm_ic_register.h): model a process or a status value with its associated scaler and unit.
- **Extended register** (tsm_ic_xregister.h)
- **Demand register** (tsm_ic_dregister.h)
- **Profile generic** (tsm_ic_profile.h): provides a generalized concept to store, sort and access sets of objects attributes.
- **Register table** (tsm_ic_rprofile.h)
- **Sensor manager**
- **Register activation**
- **Compact data** (tsm_ic_compact_data.h): compact data buffer.

11.3.3 Classes for event bound control

- **Script table**
- **Schedule**
- **Single action schedule**
- **Activity calendar**
- **Register monitor**
- **Limiter**

11.3.4 Classes for time and utilities

- **Status mapping.**
- **Special days table.**
- **Disconnect control** (tsm_ic_disc_ctrl.h): manage an internal or external disconnect unit of the meter (e.g. gas electrovalve).
- **Utility table**
- **Clock** (tsm_ic_clock.h): model the device clock.

11.3.5 Classes for setting up data exchange via local ports and modems

- **IEC local port setup**
- **Auto answer**
- **IEC twisted pair setup**
- **Auto connect**
- **Modem setup**

11.3.6 Classes for setting up data exchange over the internet

- **GPRS modem setup.**
- **MAC address setup**
- **PPP setup**
- **HDLC setup**
- **TCP/UDP setup**
- **IPV4 setup**
- **SMTP setup**

11.3.7 Classes for setting up data exchange via M-Bus

- **MBUS slave port setup**
- **MBUS client port setup**
- **MBUS client**

11.3.8 Other classes

- **Push setup** (tsm_ic_push_conf.h): push service configurations.

12 STACK PROTOCOLS

12.1 Definitions

A protocol is implemented as a stack layer and exposes to the user one or more services.

A **service** is composed of two or four primitives and each primitive identify the stage of the service execution:

- **request**: an entity requests a service,
- **indication**: an entity is notified for a request.
- **response**: an entity responds to a request that require a confirmation.
- **confirm**: an entity is notified for a confirmation.

A typical service comes in two flavors: confirmed and unconfirmed.

- **confirmed service** is a kind of service where to each request corresponds a confirmation. This service is composed by all the four primitives.
- **unconfirmed service** is a kind of service where the request should not be confirmed. This service is composed just by the request and the indication primitives.

Looking at the figure below the "*ORIGIN*" requests to the layer "*MODULE*" the service "*SVC*. This `SVC.request`, traverses the stack in the downward direction, arrives to the "*DESTINATION*" and enters his stack from the lower layer. After traversing the stack upward the *`SVC.request`* results into a *`SVC.indication`* to notify the upper entity.

If the service is an unconfirmed service the message journey terminates here otherwise a `SVC.response` is generated by the "*DESTINATION*" and is sent downward in the stack. After traversed the "*ORIGIN*" stack in the upward direction, the *`SVC.response`* results into a *`SVC.confirm`* primitive to notify the "*ORIGIN*" about the result of the request.

In a nutshell a *`SVC.request`* becomes a *`SVC.indication`* and a *`SVC.response`* becomes a *`SVC.confirm`*.

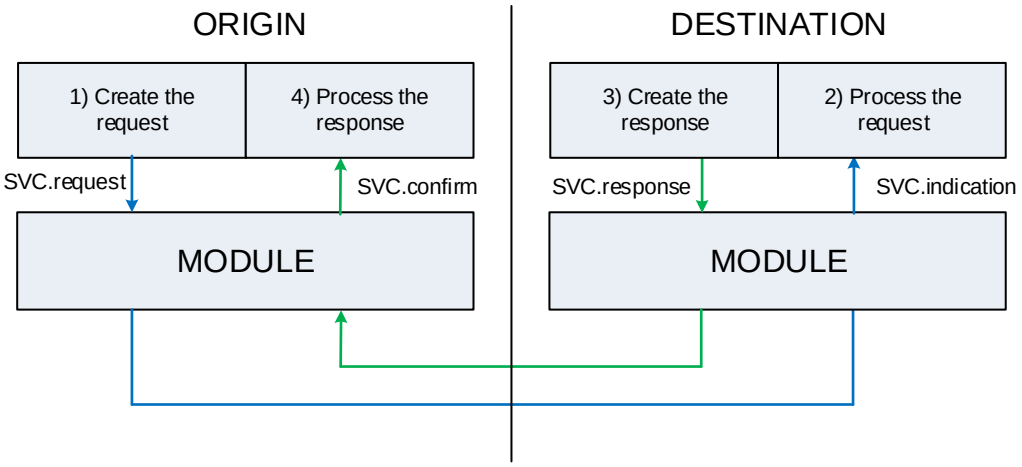


Figure 4 - service primitives

A stack protocol exposes a set of services to the user. These services are provided as functions in the form of requests and responses. The requests/responses parameters are packed in a raw string of bytes called protocol data unit (PDU) that is the unit of data exchanged between two layers at the same level and that is passed to the lower layer.

When a PDU is received from the lower layer the raw PDU is unpacked and the original service request/response parameters are recovered and the user is notified by using the service indication/confirmation. To notify the user about an event the protocol layer jumps into the user code by using a callback function call (late binding).

E.g. When a connection is established the module notify the user about the event by using the connect confirmation function. It will be a user duty to implement this confirmation function if he is interested in receiving the event notification.

As an example consider some of the services provided by the HDLC module.

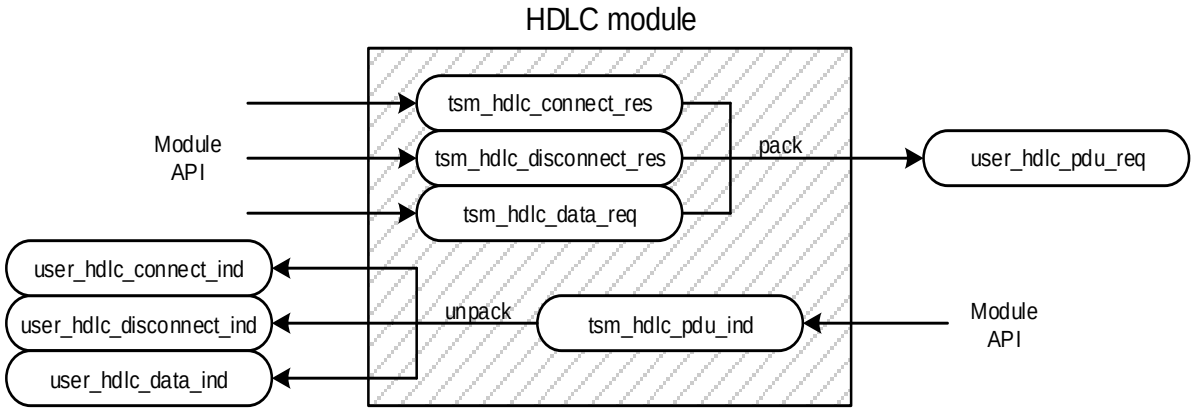


Figure 5 - module implementation

All the library modules are completely disjointed and you can easily link two modules to form the protocols stack by forwarding the pdu request of an higher module to the data request of a lower one and by forwarding the data indication of a lower module to the pdu indication of a the higher one.

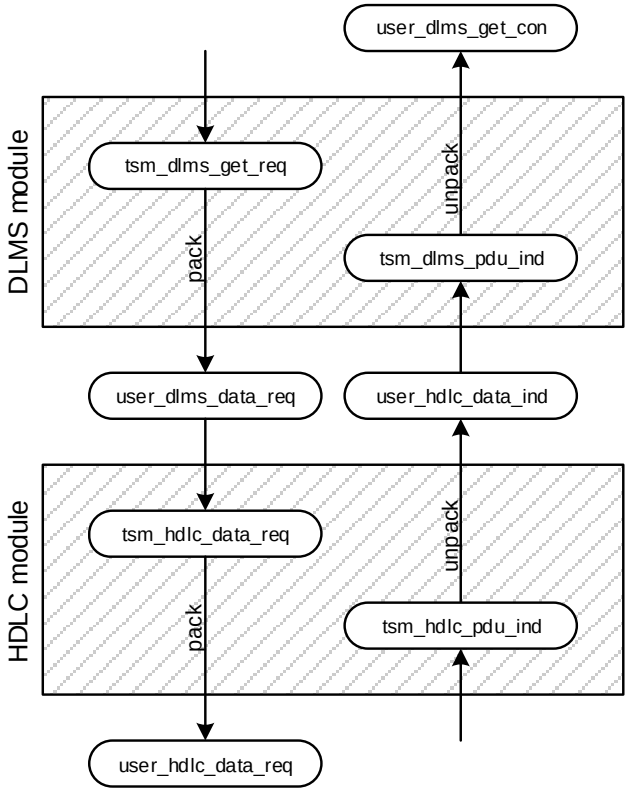


Figure 6 - modules linking

12.2 Provided protocols

The library is composed of various predefined “protocols” that are the implementations of the most common ones used in the context of Smart Metering:

- **DLMS/COSEM (IEC 62056-53): D**evice Language Message Specification **a**pplication layer.
- **SML (IEC 62056-58):** Smart Meter Language application layer.
- **MBUS-AL (EN 13757-3):** Meter BUS application layer.
- **HDLC (IEC 62056-46):** High Level Data Link Control layer. Mainly used over serial communications.
- **WRAP (IEC 62056-47):** a lightweight and almost stateless layer. Provided to be used as an additional multiplexing/demultiplexing protocol.

[omissis]

13 ARCHITECTURE SUMMARY

This chapter shows how all the exposed concepts and the user code fit together to create the final application architecture. In particular we will show how to join some of the library pre-defined modules to build the standard **DLMS, SML and MBUS** stacks.

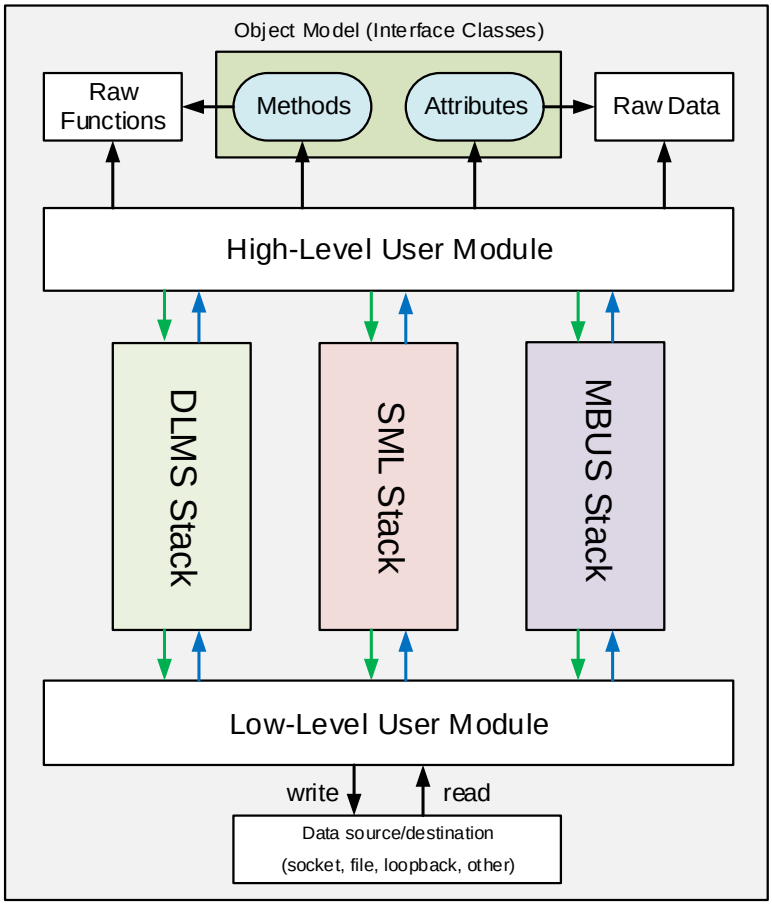


Figure 7 – modules stack and architecture

At the lowest layer of the stack we find the low level user module. This part of application has the responsibility to read and write raw data from a user defined source/destination. A data read starts a chain of data.indication upcalls in the stack and a data write is called from above via a data.request.

At the highest layer of the stack we find the high level user. Server side, when a get/set/execute indication is received from below **the raw data and functions are accessed with or without the help of the object model.**

Client side, the high level user provides the get/set/act confirm handling routines. That is, the client invokes one of the lower level get/set/execute request service primitive and asynchronously receives a response via the relative confirmation.

The standard DLMS stack is composed by the DLMS/COSEM application layer module above the HDLC or the TCPW module.

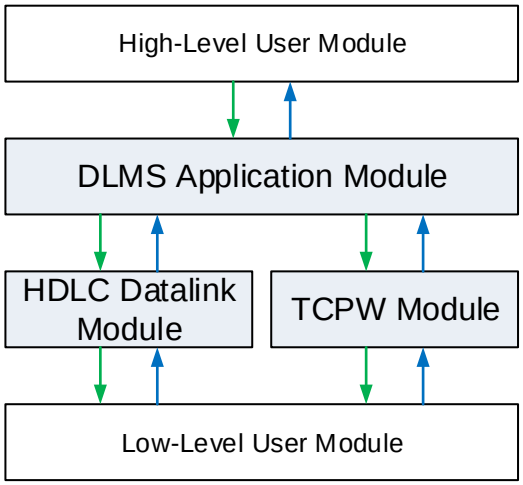


Figure 8 - DLMS stack configuration

The SML stack is composed of the SML application layer module above a user provided TLS module.

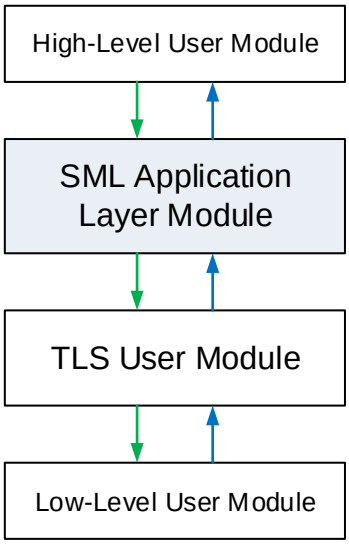


Figure 9 - SML stack configuration

14 DLMS/COSEM PROTOCOL DETAILS

This chapter deals to the issues usually associated with the DLMS/COSEM application protocol standard as provided by the DLMS UA.

14.1 Standard stack

Although the TSM library permit to place the DLMS Application layer module over an arbitrary module, the standard suggests two precise stack options.

The first option is thought to be used in a local serial communication:

- Application level: DLMS/COSEM
- Data link level: HDLC
- Physical level: serial line

The second option is thought to be used in communications over TCP/IP:

- Application level: DLMS/COSEM
- Middle level: TCPW
- Transport level: TCP
- Network level : IP
- Lower lever: whatever
-

[omissis]

15 SML PROTOCOL DETAILS

[omissis]

16 MISRA C: 2012 - COMPLIANCE MATRIX (TO FINISH)

[omissis]

17 FREQUENTLY ASKED QUESTIONS

1. How the library is supplied?
   The library is provided as a static archive of object files or as a dynamic reallocable module depending on the user requirements.

2. What is provided with the library?
   Along with the library we provide the API header files, a *Doxygen* API reference, a technical user manual explaining how the library works along with some use case examples and the sources of a simple demo server project usable as a starting point for your server side project.

3. Is the library bound to a particular **operating system**?
   No, the library is created to be portable and self-contained. There are no external dependencies.

4. In the library bound to a particular **machine architecture**?
   As long as there is a C compiler compliant with the ANSI-C standard the library can be compiled for your architecture. The library has already been tested with: generic ARM, Cortex-M3, Texas Instruments MSP-430, Renesas 78K0, Intel x86, Intel x64, Atmel AVR and MIPS machines.

5. The library is compliant with the DLMS standard?
   The library implements the DLMS/COSEM protocol respecting the standard defined by the DLMS UA colored books. Is a user duty to implement a final application that is a final application that is coherent with the requirements of a particular country normative.

6. The library is compliant with the SML standard?
   The library implements the SML protocol respecting the standard defined by the SML papers for the German market. Is a user duty to implement a final application that is coherent with the requirements of a particular country normative.

7. Can I define my own custom objects ?
   The Library do not set any limit to the custom objects. You can use the provided standard classes or you can define your custom ones to create the object instances. Just pay attention that your objects instance identifiers do not overlap with the standard ones.

17.1 License and conditions

Terranova TSM Library is offered to the market with the following commercial policies:

- The library is supplied with a price per unit.
- The prices decrease with the increasing number of bought libraries according to the price range
- There are no developing cost to pay.
- Terranova ensures the library will be supplied ready to be embedded in the devices.
- For further information and/or a dedicated quote please contact Terranova.

Sales Department at sales@terranovasoftware.eu

17.2 Known problems

There are no known issues.

17.3 Support

Please send support requests and bug reports to tsml@terranovasoftware.eu

17.4 Changelogs

The changelogs ships within the Doxygen API Documentation.