EPRI DLMS/COSEM
Open Source Reference Design

Developer and Maintainer Guide

Prepared by Gregory Barrett, Otter Peak Engineering, LLC

12/30/2016

Contents

Code Map 3
Architectural Overview 3
Server Application 6
Client Application 16
Core Classes 19
DLMSVector 19
DLMSOptional 20
DLMSVariantInitList 20
DLMSBitSet 20
DLMSVariant 20
DLMSSequence 20
DLMSValue 20

Code Map

Top Level	
[lib]	This folder contains the source code for the library as well as source code for other required libraries.
[src]	This folder contains the source code for example applications that use the library. Current example implementations are for Linux and a Cortex-M3 development board from ST Micro.
[test]	This folder contains the source code for library unit tests.
CMakeLists.txt	The cmake list file for the building the entire project.
DLMS-COSEM.sln	The Microsoft Visual Studio solution file for building the entire project using VisualGDB.
README	Standard README file.
README.md	github README file.
doxyfile	The Doxygen configuration file for building library documentation.

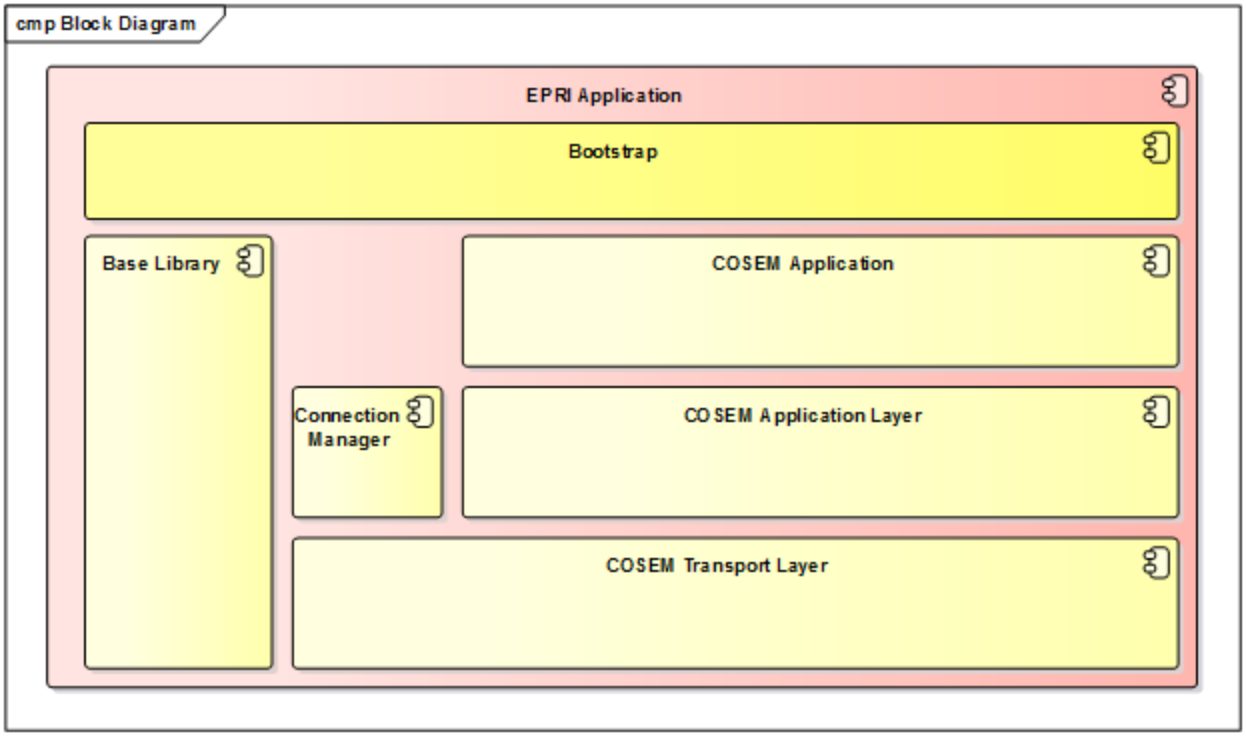
[lib]	
[DLMS-COSEM]	The Linux version of the library. Output is a static library, *libDLMS-COSEM.a*.
[STM32-DLMS-COSEM]	The STM32 version of the library. Output is a static library, *libSTM32-DLMS-COSEM.a*.
[asio-1.10.6]	The ASIO library is used in the Linux implementation to provide serial and TCP access.
[googletest-release-1.7.0]	Unit tests for the library utilize Google Test.
CMakeLists.txt	The cmake list file for building the libraries.

[src]	
[Linux]	The Linux example application which supports both client and server.
[STM32-NUCLEO-F207ZG]	The STM32 example application which supports server. Client could be easily supported, but server is the most appropriate.
CMakeLists.txt	The cmake list file for building the examples.

[test]	
[DLMS-COSEM-TEST]	The Linux unit tests for the library. These tests utilize the Google Test framework.
CMakeLists.txt	The cmake list file for building the unit tests.

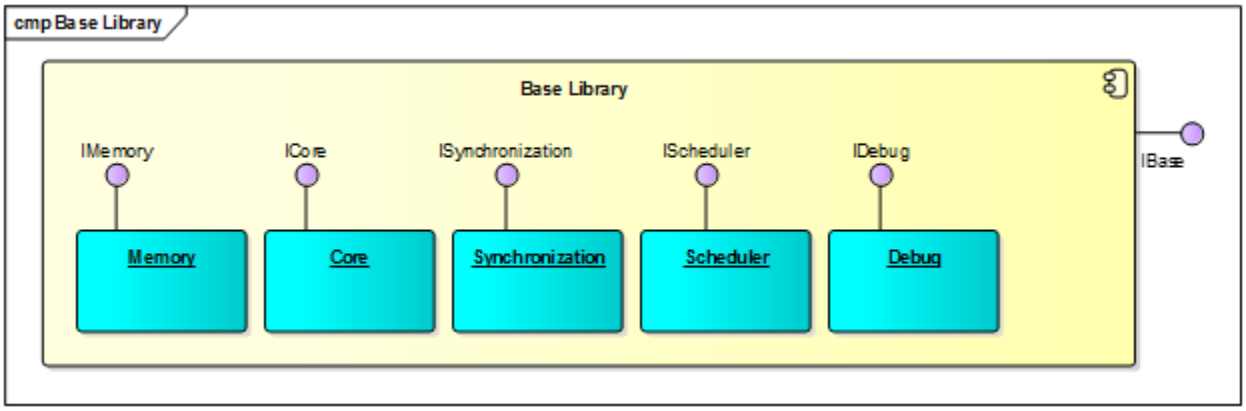
Architectural Overview

The architecture of the library follows the reference detailed in the DLMS/COSEM Green Book.

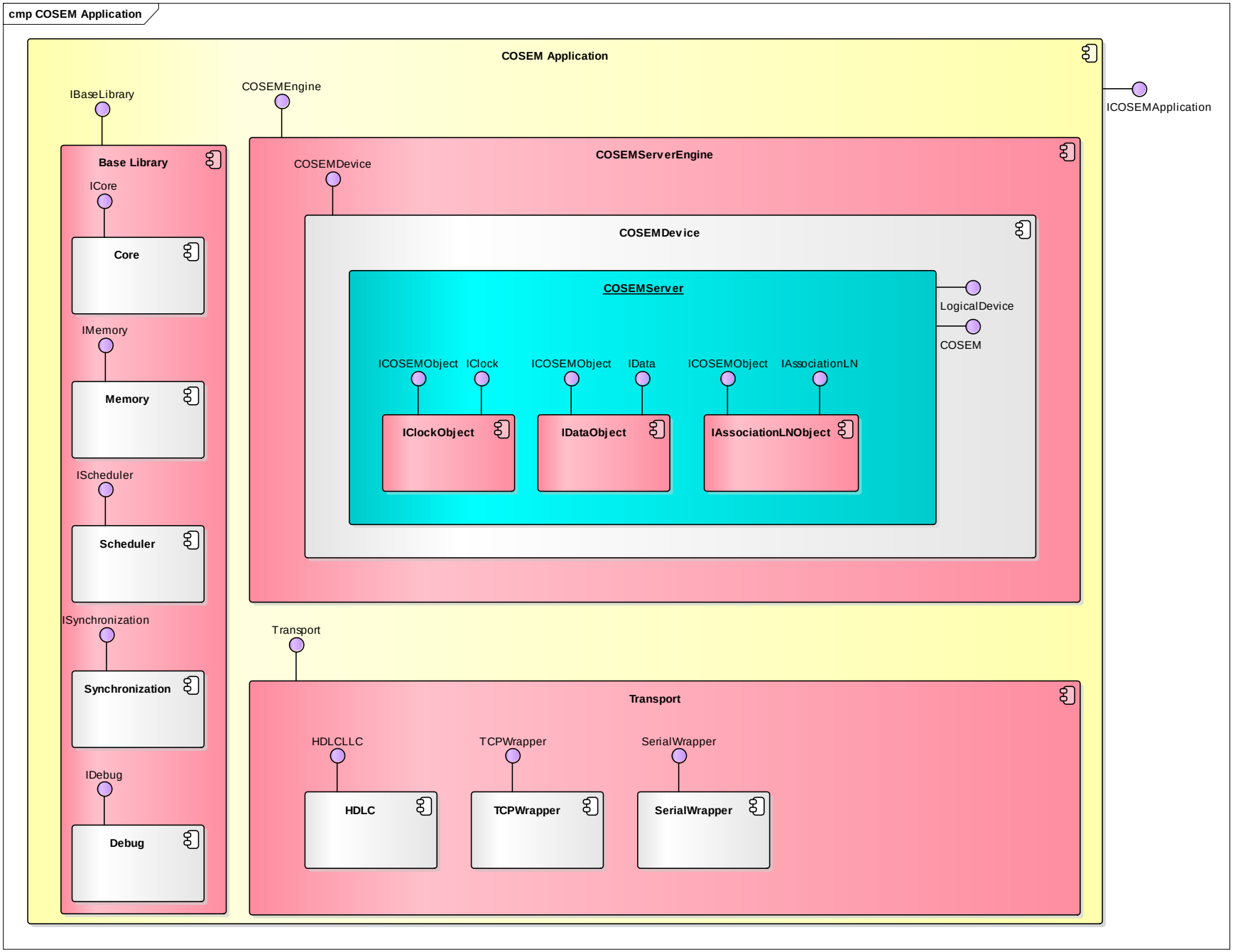


The core DLMS/COSEM library ("library") has been designed to be portable if using a *gcc*-based compiler. Porting is performed through implementation of a set of interfaces within the Base Library. These interfaces provide functionality like scheduling, memory allocation, and communication abstractions. Example "ports" for Linux and the STM32 are provided, but are not "hardened" implementations at this point in the development cycle.

The Linux application is built using cmake. The STM32 application is built using make.



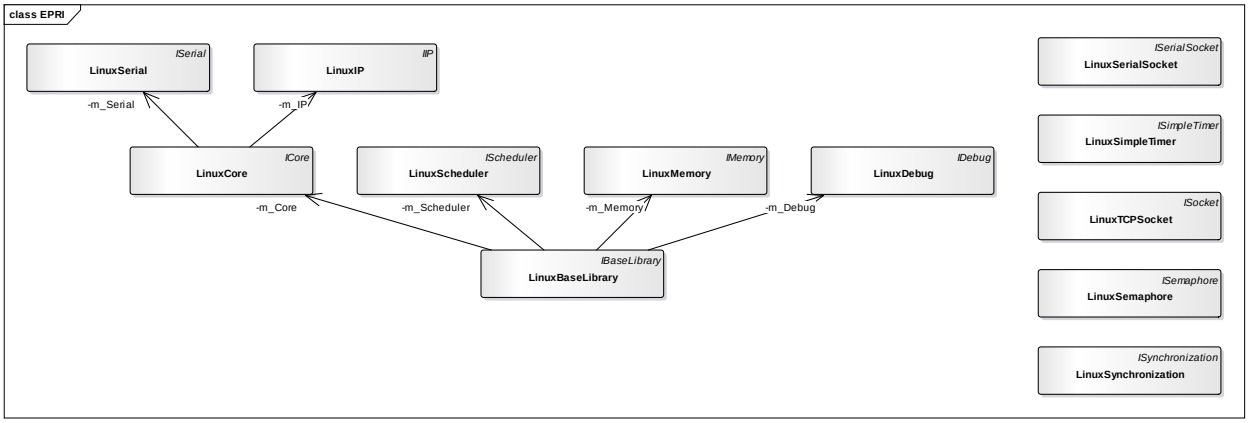
An example implementation of the Base Library can be found in the [src/Linux] folder.


cmp COSEM Application
COSEM Application
IBaseLibrary
COSEMEngine
ICOSEMApplication
Base Library
ICore
Core
IMemory
Memory
IScheduler
Scheduler
ISynchronization
Synchronization
IDebug
Debug
COSEMServerEngine
COSEMDevice
COSEMDevice
COSEMServer
LogicalDevice
COSEM
ICOSEMObject
IClock
IClockObject
ICOSEMObject
IData
IDataObject
ICOSEMObject
IAssociationLN
IAssociationLNObject
Transport
Transport
HDLCLLC
HDLC
TCPWrapper
TCPWrapper
SerialWrapper
SerialWrapper

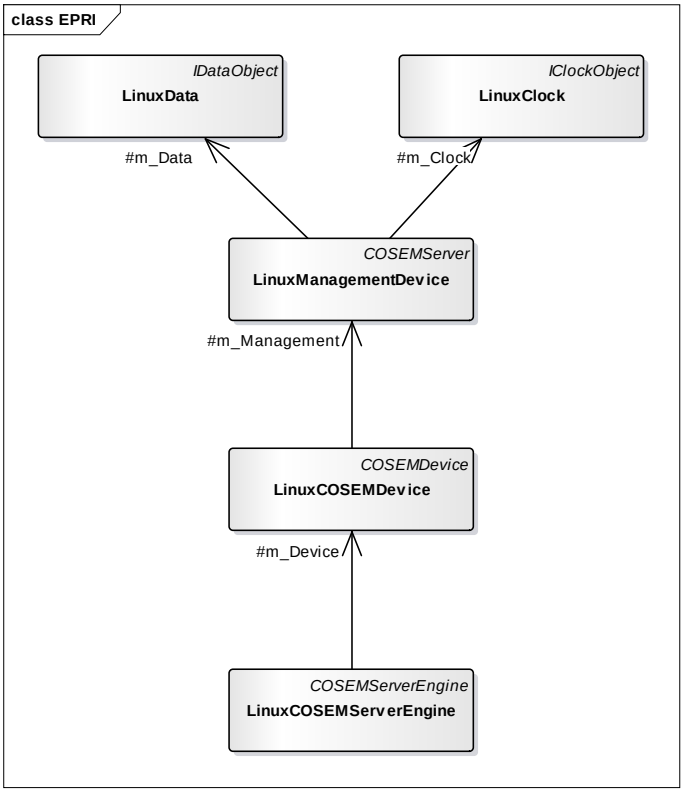
Server Application

A COSEM Server Application is comprised of:

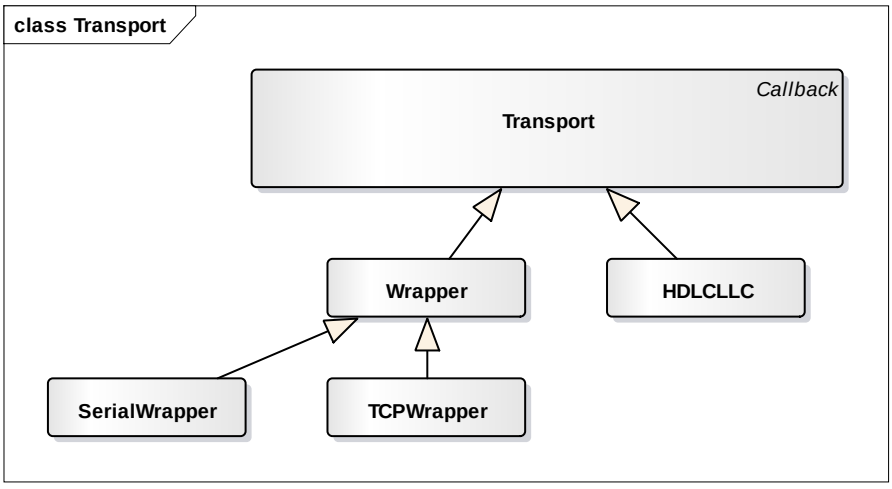
- an implementation and instantiation of the Base Library,



- an instance of the COSEMServerEngine which contain one or more
    - instances of COSEMDevice (physical device) which contain one or more
        - instances of COSEMServer (logical device) which contain one or more
            - instance of COSEMObject (COSEM object)



- an instance of one or more transports:



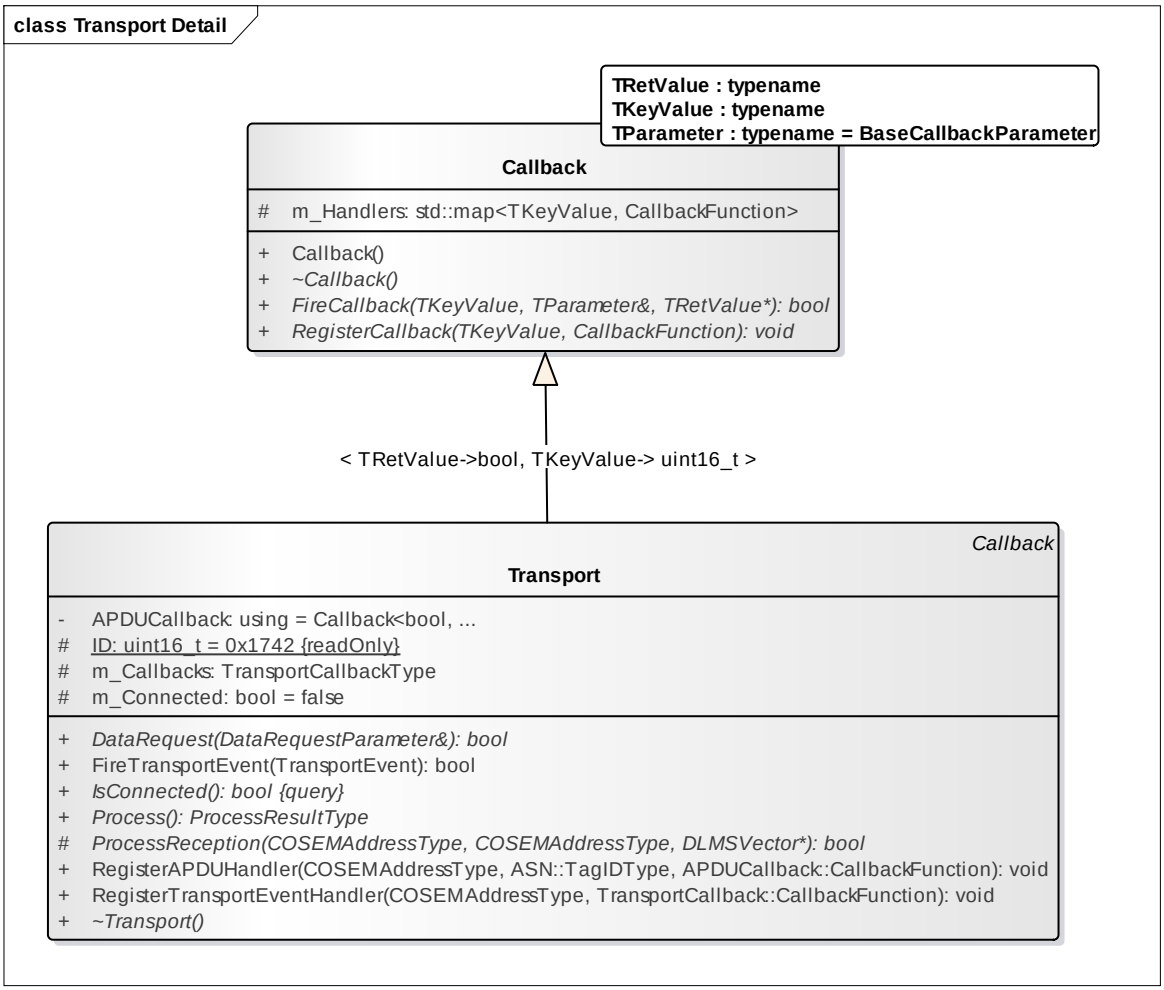
In the example Linux code, creation of a COSEM TCP server looks like the following:

```
m_pServerEngine = new LinuxCOSEMServerEngine(COSEMServerEngine::Options(),
    new TCPWrapper((pSocket = Base()->GetCore()->GetIP()->CreateSocket(LinuxIP::Options()))));
if (SUCCESSFUL != pSocket->Open())
{
    PrintLine("Failed to initiate listen\n");
}
```

The COSEMServerEngine needs to be associated with a Transport which needs to be associated with a physical means to get data from point A to point B.

Transport is the interface to all lower communication layers. Incoming byte streams are processed by derived Transport classes, resulting in COSEM APDU objects. Transport is derived from the Callback library class. This provides a standard mechanism for callers to register callbacks when certain events occur.

Transport also provides the upper layers with a common means to determine (and handle) connection and disconnection of the physical medium. This is done through the same Callback mechanism described above. Two "events", TRANSPORT_CONNECTED and TRANSPORT_DISCONNECTED.



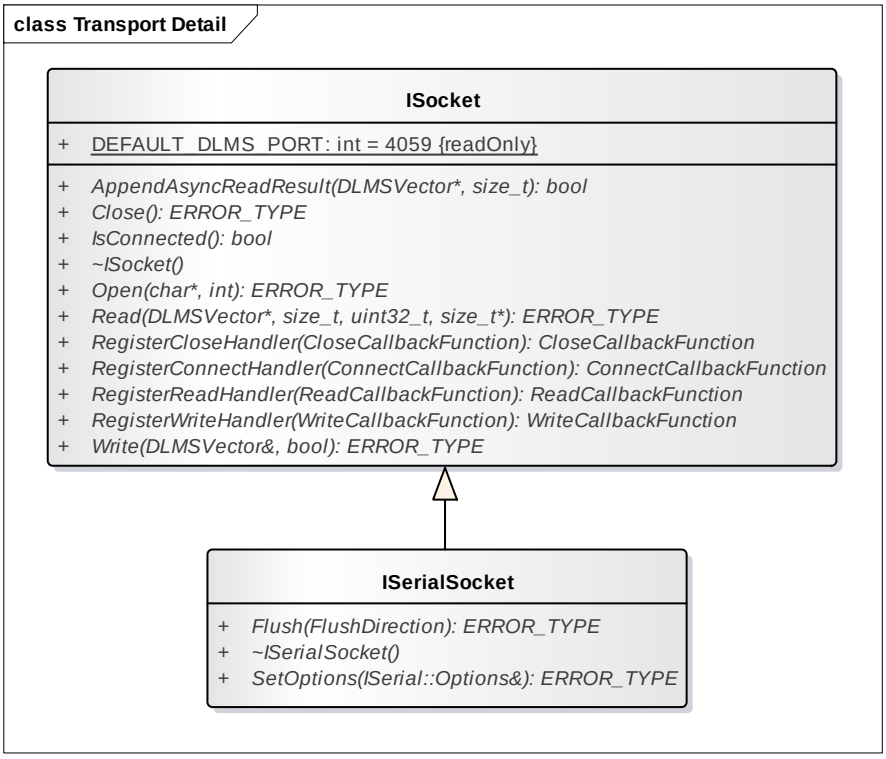
Callback is a template class. It uses a key/callback function pair to allow registration of functions by a unique identifier. There are many examples within the library of this class.

```
//
// Packet Handlers
//
m_PacketCallback.RegisterCallback(HDLCControl::INFO,
    std::bind(&HDLCMAC::I_Handler, this, std::placeholders::_1));
```

.
.
.

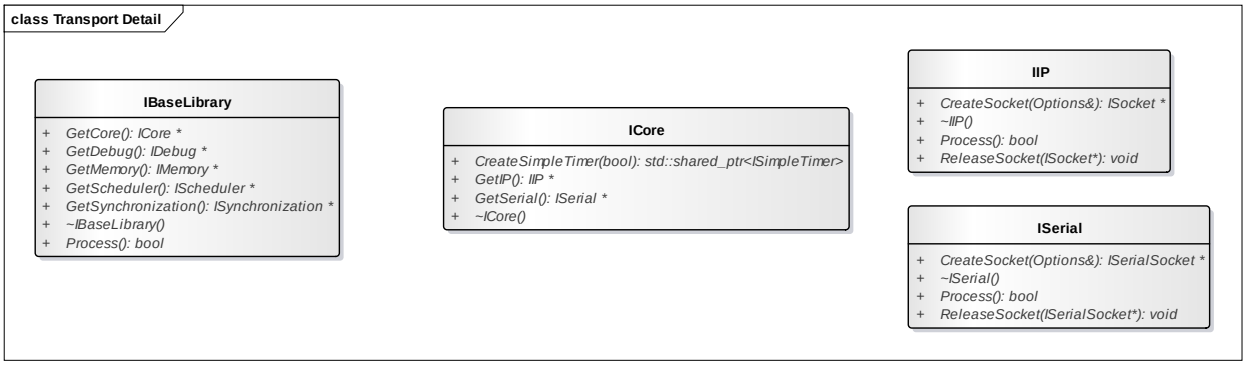
```
bool CallbackRetVal = false;
if (m_PacketCallback.FireCallback(PacketType, *pRXPacket, &CallbackRetVal) &&
    !CallbackRetVal)
```

All physical communication is represented through a single base interface, *ISocket*.

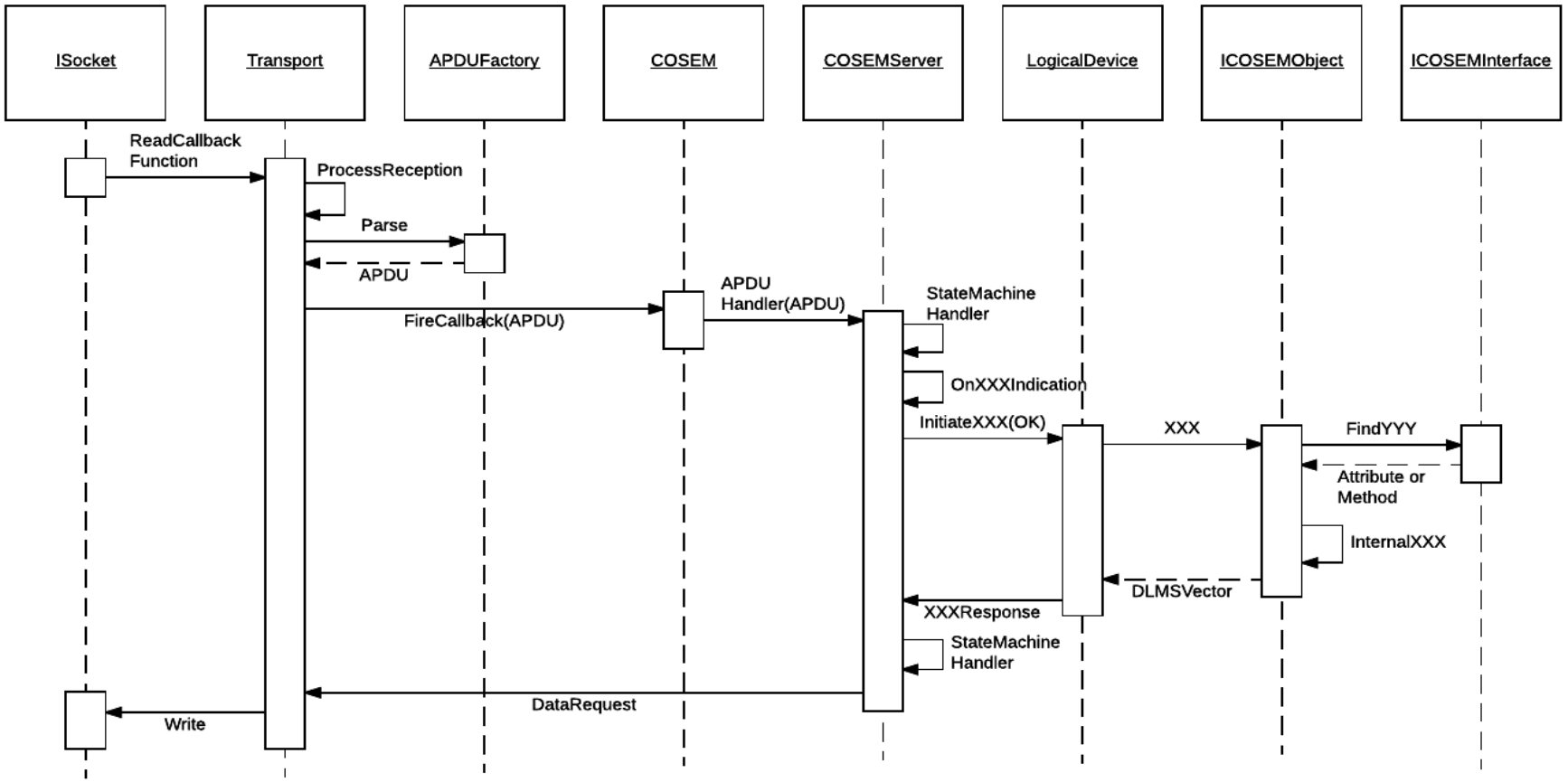


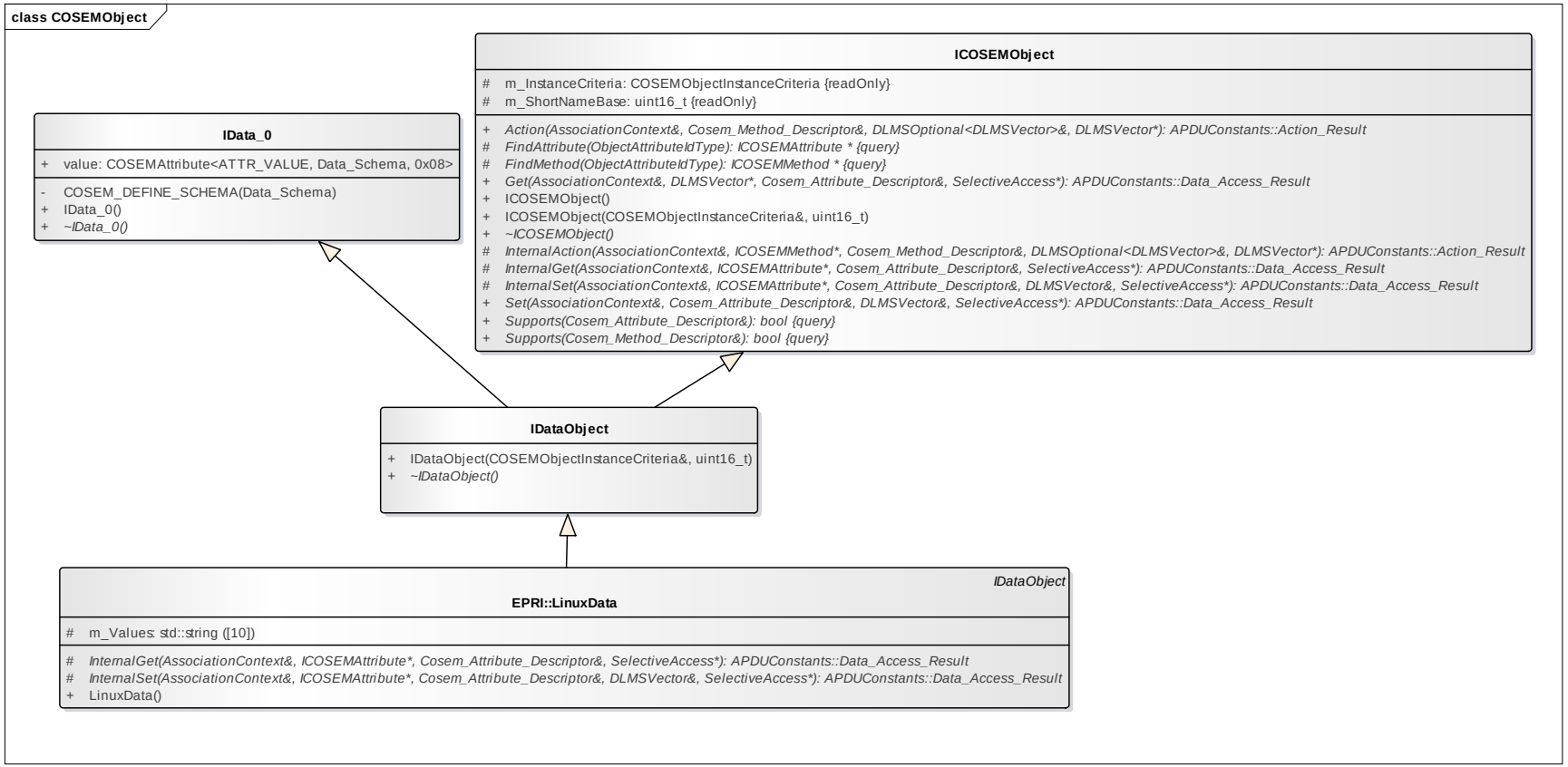
This abstraction allows for asynchronous and synchronous reading and writing of a physical connection. In the library, this could be a serial or a TCP connection. The callbacks that can be registered through ISocket are used by the Transport implementations to inform the upper layers of changes in state and/or the arrival of data.

Sockets are created through appropriate Base Library classes.



The following sequence diagram gives an overview of the data flow from the server point of view. Bytes flow from ISocket to be parsed by the Transport. The appropriate APDU object is created and delivered to the appropriate handler where it is processed by the LogicalDevice and ICOSEMObjects.


ISocket
Transport
APDUFactory
COSEM
COSEMServer
LogicalDevice
ICOSEMObject
ICOSEMInterface
ReadCallback Function
ProcessReception
Parse
APDU
APDU Handler(APDU)
FireCallback(APDU)
StateMachine Handler
OnXXXIndication
InitiateXXX(OK)
XXX
FindYYY
Attribute or Method
InternalXXX
DLMSVector
XXXResponse
StateMachine Handler
DataRequest
Write



The class diagram above illustrates the classes involved in implementing a COSEM object for your server. The COSEM interface is defined by the library. You need to implement the appropriate virtual functions to handle client requests. As an example, the following code with explanation should give you a general idea of the steps necessary.

```cpp
//
// Data
//
LinuxData::LinuxData()
    : IDataObject({ 0, 0, 96, 1, {0, 9}, 255 })
{
    for (int Index = 0; Index < 10; ++Index)
    {
        m_Values[Index] = "LINUXDATA" + std::to_string(Index);
    }
}

APDUConstants::Data_Access_Result LinuxData::InternalGet(const AssociationContext& Context,
    ICOSEMAttribute * pAttribute,
    const Cosem_Attribute_Descriptor& Descriptor,
    SelectiveAccess * pSelectiveAccess)
{
    pAttribute->SelectChoice(COSEMDataType::VISIBLE_STRING);
    pAttribute->Append(m_Values[Descriptor.instance_id.GetValueGroup(EPRI::COSEMObjectInstanceID::VALUE_GROUP_E)]);
    return APDUConstants::Data_Access_Result::success;
}

APDUConstants::Data_Access_Result LinuxData::InternalSet(const AssociationContext& Context,
    ICOSEMAttribute * pAttribute,
    const Cosem_Attribute_Descriptor& Descriptor,
    const DLMSVector& Data,
    SelectiveAccess * pSelectiveAccess)
{
    APDUConstants::Data_Access_Result RetVal = APDUConstants::Data_Access_Result::temporary_failure;
    try
    {
        DLMSValue Value;

        RetVal = ICOSEMObject::InternalSet(Context, pAttribute, Descriptor, Data, pSelectiveAccess);
        if (APDUConstants::Data_Access_Result::success == RetVal &&
            pAttribute->GetNextValue(&Value) == COSEMType::GetNextResult::VALUE_RETRIEVED)
        {
            m_Values[Descriptor.instance_id.GetValueGroup(EPRI::COSEMObjectInstanceID::VALUE_GROUP_E)] =
                DLMSValueGet<std::string>(Value);
            RetVal = APDUConstants::Data_Access_Result::success;
        }
        else
        {
            RetVal = APDUConstants::Data_Access_Result::type_unmatched;
        }
    }
```

Defines the OBIS Criteria this Object Will Handle

0.0.96.1.[0-9].255

GET Service

Build the Response via COSEMType

SET Service

InternalSet parses the incoming DLMSVector and makes it available for manipulation if necessary.

All data structures defined by COSEM are through a single recursive type. The library represents this through the COSEMType class. COSEMObject attributes, for instance, are derived from COSEMType.

A COSEMType is defined through a schema. This allows the server (and potentially the client) to know exactly what information is being transferred through an interface. The DLMS/COSEM Blue Book defines the standard set of interfaces available to developers. The following will take you through an example using Class 15 to give you a feel for the process you would use to define your own classes.

Association LN		0...MaxNbofAss.	class_id = 15, version = 1			
Attributes		*Data type*	*Min.*	*Max.*	*Def.*	Short name
1. logical_name	(static)	octet-string				x
2. object_list	(static)	object_list_type				x + 0x08
3. associated_partners_id		associated_partners_type				x + 0x10
4. application_context_name		application-context-name				x + 0x18
5. xDLMS_context_info		xDLMS-context-type				x + 0x20
6. authentication_mechanism_ name		mechanism-name				x + 0x28
7. LLS_secret		octet-string				x + 0x30
8. association_status		enum				x + 0x38
9. security_setup_reference	(static)	octet-string				x + 0x40
Specific methods		*m/o*				
1. reply_to_HLS_authentication (data)		o				x + x60
2. change_HLS_secret (data)		o				x + x0x68
3. add_object (data)		o				x + 0x70
4. remove_object (data)		o				x + 0x78

We will use the ***xDLMS_context_info*** attribute as our example:

xDLMS_context_info Contains all the necessary information on the xDLMS context for the given association.

xDLMS-context-type::= structure

{
conformance: bitstring(24),
max_receive_pdu_size: long-unsigned,
max_send_pdu_size: long-unsigned,
dlms_version_number: unsigned,
quality_of_service: integer,
cyphering_info: octet-string
}

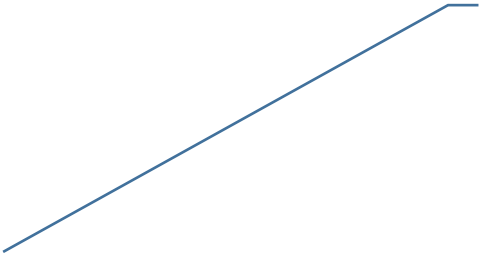
The following schema defines this attribute (.cpp):

```
COSEM_BEGIN_SCHEMA(IAssociationLN_0::xDLMS_Schema)
    COSEM_BEGIN_STRUCTURE
        COSEM_BIT_STRING_TYPE
        COSEM_LONG_UNSIGNED_TYPE
        COSEM_LONG_UNSIGNED_TYPE
        COSEM_UNSIGNED_TYPE
        COSEM_INTEGER_TYPE
        COSEM_OCTET_STRING_TYPE
    COSEM_END_STRUCTURE
COSEM_END_SCHEMA
```

```
    class IAssociationLN_0 : public ICOSEMInterface
    {
.
.
.
        COSEM_DEFINE_SCHEMA(xDLMS_Schema)

    public :

        enum Attributes : ObjectAttributeIdType
        {
.
.
.
            ATTR_XDLMS_CON_INFO      = 5,
.
.
.
        };
.
.
.
        COSEMAttribute<ATTR_XDLMS_CON_INFO, xDLMS_Schema, 0x20>          xDLMS_context_type;
```



Once the definition is in place, the developer can utilize the methods of COSEMType to manipulate:

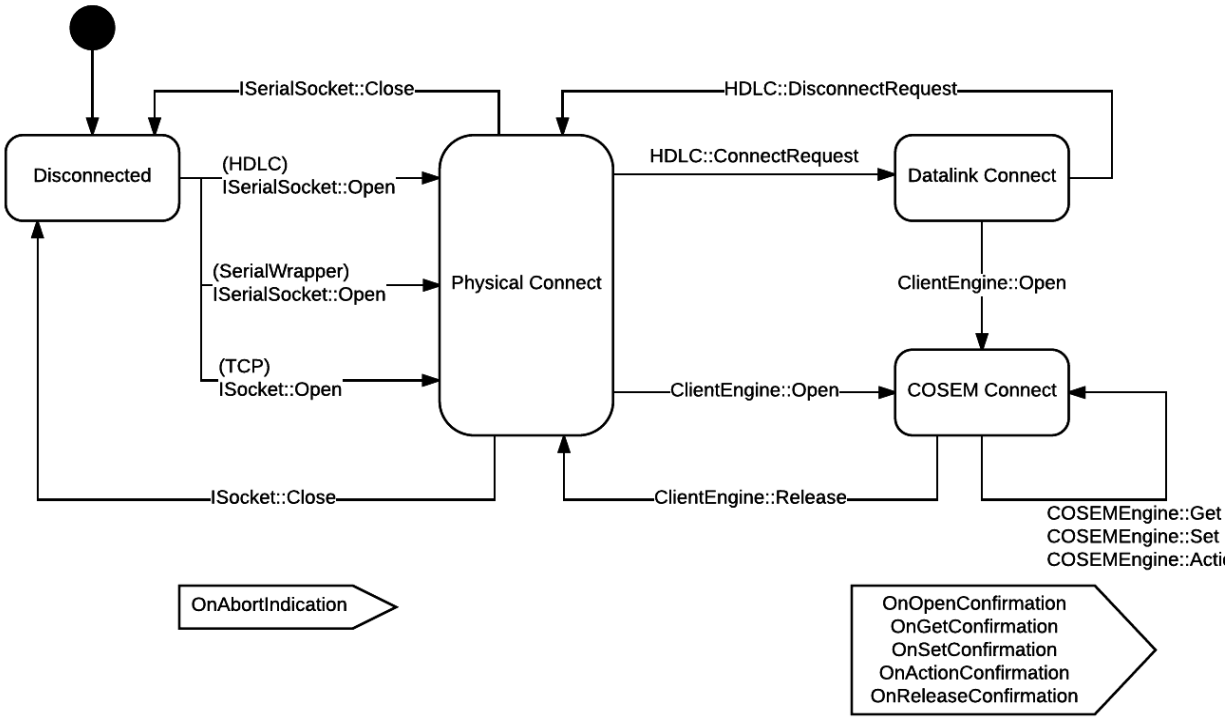
```
    APDUConstants::Data_Access_Result Association::InternalGet(const AssociationContext& Context,
        ICOSEMAttribute * pAttribute,
        const Cosem_Attribute_Descriptor& Descriptor,
        SelectiveAccess * pSelectiveAccess)
    {
.
.
.
        case ATTR_XDLMS_CON_INFO:
            AppendResult = pAttribute->Append(
                DLMSSequence
                ({
                    pContext->m_xDLMS.ConformanceBits(),
                    pContext->m_xDLMS.APDUSize(),
                    pContext->m_xDLMS.APDUSize(),
                    pContext->m_xDLMS.DLMSVersion(),
                    pContext->m_xDLMS.QOS(),
                    pContext->m_xDLMS.DedicatedKey()
                }));
            break;
```

The library provides a single variant type called DLMSValue to act as the container for COSEM data. This data type allows for both an array (DLMSSequence) or a single value (DLMSVariant) to be stored in the same variable. In the example above, we are creating a sequence of elements to represent the values contained with the structure. The library will validate that all elements are present and can be converted to the schema types. If not, an error will be returned.

This design pattern is used throughout the library. A similar class, ASNType, provides support for parsing of APDUs.

Client Application

The client application uses the same Base Library interfaces as the server.



The diagram above should give you a good indication of the normal flow of a client application. Notice the different paths for datalink-based protocols such as HDLC. Once in the “COSEM Connect” state, interfacing to the library is the same regardless of the transport mechanism.

Confirmations for requests are provided via callbacks, OnXXXConfirmation. Developers can use these callbacks to process responses as in the following example:

```
virtual bool OnGetConfirmation(RequestToken Token, const GetResponse& Response)
{
    Base()->GetDebug()->TRACE("\n\nGet Confirmation for Token %d...\n", Token);
    if (Response.ResultValid && Response.Result.which() == Get_Data_Result_Choice::data_access_result)
    {
        Base()->GetDebug()->TRACE("\tReturned Error Code %d...\n",
            Response.Result.get<APDUConstants::Data_Access_Result>());
        return false;
    }

    if (CLSID_IData == Response.Descriptor.class_id)
    {
        IData     SerialNumbers;
        DLMSValue Value;

        SerialNumbers.value = Response.Result.get<DLMSVector>();
        if (COSEMType::VALUE_RETRIEVED == SerialNumbers.value.GetNextValue(&Value))
        {
            Base()->GetDebug()->TRACE("%s\n", DLMSValueGet<VISIBLE_STRING_CType>(Value).c_str());
        }

    }
    else if (CLSID_IAssociationLN == Response.Descriptor.class_id)
    {
        IAssociationLN CurrentAssociation;
        DLMSValue      Value;

        switch (Response.Descriptor.attribute_id)
        {
        case IAssociationLN::ATTR_PARTNERS_ID:
            {
                CurrentAssociation.associated_partners_id = Response.Result.get<DLMSVector>();
                if (COSEMType::VALUE_RETRIEVED == CurrentAssociation.associated_partners_id.GetNextValue(&Value) &&
                    IsSequence(Value))
                {
                    DLMSSequence& Element = DLMSValueGetSequence(Value);
                    Base()->GetDebug()->TRACE("ClientSAP %d; ServerSAP %d\n",
                        DLMSValueGet<INTEGER_CType>(Element[0]),
                        DLMSValueGet<LONG_UNSIGNED_CType>(Element[1]));
                }
            }
            break;

        default:
            Base()->GetDebug()->TRACE("Attribute %d not supported for parsing.", Response.Descriptor.attribute_id);
            break;
        }
    }
```

An Example of Parsing a GET Response.

Use the appropriate interface and attributes to get the data.

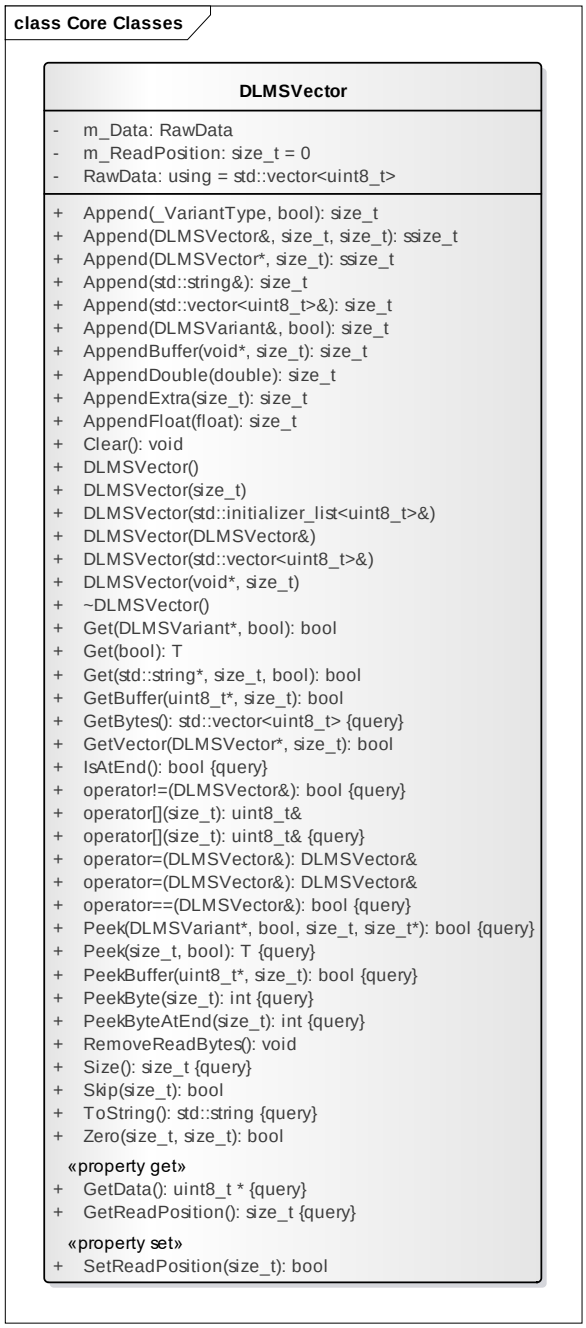
Core Classes

There are several classes that are used throughout the project and deserve a short mention.

DLMSVector

This class ultimately contains a std::vector<uint8_t>. It provides operations to help convert for endianness and other helpful operators.

General usage is simple with AppendXXX, GetXXX, and PeekXXX functions for different base types. Read position is kept separate from append allowing for consumers and producers to operate separately.



DLMSOptional

```
template <typename T>
    using DLMSOptional = std::experimental::optional<T>;
#define DLMSOptionalNone std::experimental::nullopt
```

DLMSVariantInitList

```
using DLMSVariantInitList = std::initializer_list<uint32_t>;
```

DLMSBitSet

```
using DLMSBitSet = std::bitset<64>;
```

DLMSVariant

```
using DLMSVariant =
   variant<blank, bool, int8_t, uint8_t, int16_t, uint16_t, int32_t, uint32_t,
           int64_t, uint64_t, std::string, float, double, DLMSVector,
           DLMSVariantInitList, DLMSBitSet>;
```

DLMSSequence

```
using DLMSSequence = std::vector<DLMSVariant>;
```

DLMSValue

```
using DLMSValue = variant<DLMSVariant, DLMSSequence>;
```