Open source solutions for smart metering

Home

Read basic information from Gurux.DLMS component before you start to create DLMS client.

Simple meter reading example

Before use the following device parameters must be set. Parameters are manufacturer specific

- Is Short or Logical name used.
- Server address
- Client address
- Interface type

Usually you can use 1 as Server address and 16 as Client address when authentication is not used. If authentication is used Client address changes. You can read here to find out how Client and Server addresses are counted.

Some manufacturers might use own custom Server and Client addresses.
Note!
If Server address is wrong meter does not send response. If Client address is wrong you will get authentication error.

Using authentication

When authentication (Access security) is used server(meter) can allow different rights to the client. Without authentication (None) only read is allowed. Gurux DLMS components supports six different authentication level:

- None
- Low
- High
- HighMD5
- HighSHA1
- GMAC

Java | C# | Delphi | ANSI C++ | ANSI C | Python

```
GXDLMSClient client = new GXDLMSClient();

// Is used Logican Name or Short Name referencing.
client.setUseLogicalNameReferencing(true);

// Is used HDLC or COSEM transport layers for IPv4 networks (IEC 62056-47)
client.setInterfaceType(InterfaceType.HDLC);
client.setClientAddress(16);
client.setServerAddress(1);
```

?

Connection Initialization

The connection initialization depends on the connection type and the device. Some devices require IEC62056-21 protocol handshake when serial port connecion is used before starting to communicate using DLMS protocol.

Serial/Modem handshake

This is done only with serial and modem connections. Direct tcp/ip doesn't need this. If IEC62056-21 handshake is used it is done before this.

This handshake uses SNMR request to gather packet and window size information from the device.

If the device does not reply to this message usually Server address is wrong or meter do not support DLMS.

Once the full reply data is received call ParseUAResponse method to set configure GXCOSEM Component with correct settings.

AARE request

This is the first command that is mandatory for all connections and device types. This command tells the device if authentication is used and whether Long Name or Short Name reference is used. The packet can be generated with AARQRequest method and it uses UseLogicalName and Authentication properties so make sure these are set to correct values.

If password is defined all data do not necessary do not fit to one message.

Once the full reply is received parse it with ParseAAREResponse method. This method sets the relevant settings to the GXCOSEM component and return a collection of manufacturer specific tags if there was any.

Now the connection is established.

Java | C# | Delphi | ANSI C++ | ANSI C | Python

```java
GXReplyData reply = new GXReplyData();
byte[] data;
data = client.SNRMRequest();
if (data != null)
{
    readDLMSPacket(data, reply);
    //Has server accepted client.
    client.parseUAResponse(reply.getData());
}

//Generate AARQ request.
//Split requests to multiple packets if needed.
//If password is used all data might not fit to one packet.
for (byte[] it : client.AARQRequest())
{
    reply.clear();
    reply = readDLMSPacket(it, reply);
}
//Parse reply.
client.parseAAREResponse(reply.getData());
```

If parameters are right connection is made. Next you can read Association view and show all objects that meter can offer.

Association View

Association View describes what kind of objects meter offers. The association view is requested using data from GetObjects method. The reply data is very large so this can take very long time (even hours with slow connection like terminal and if the connection is bad). Because this takes so long it is recommended to save this information for future use as it doesn't change unless meter software is updated. The data structure varies between manufacturers and device models.

The reply data is parsed using ParseObjects method in GXCOSEM Component. The method returns a collection of DLMS Objects.

Some of the data object may be typed as **Profile Generic**. These are special objects that have columns and rows instead of single value. You can think Profile Generic as a Table.

You should read captures objects (Column names) of Profile Generics here.

Note!
Some meters can return different objects depending from Authentication level. Example Actaris returns only Device ID with Low authentication level. Also Indian Standard IS 15959 defines what what kind of objects are available different authentication levels.

Java | C# | Delphi | ANSI C++ | ANSI C | Python

```
/// Read Association View from the meter.
GXReplyData reply = new GXReplyData();
readDataBlock(client.getObjects(), reply);
GXDLMSObjectCollection objects = client.parseObjects(reply.getData(), true);
```

Reading values

Reading is generally split in two. Reading of "regular" data objects and profile generic objects.

General Data Objects

Data objects are read using Read method. The parameters are defined as follows:

Item: COSEM Object to read.
Attribute Ordinal: The ordinal number of the requested attribute.

The data is updated using UpdateValue method that returns the value in the format that the device provided.

Profile Generic objects

There are two ways to read Profile Generic data. By Entry or Range. In entry parameters tell where read is started (Zero index) and how many items are read. In range parameters tell starting and ending time.

Note! All meters do not support read by entry or range. You can check is this supported from conformance.

The request is generated using ReadRowsByEntry or ReadRowsByRange methods.

Java | C# | Delphi | ANSI C++ | ANSI C | Python

```
//Read object.
int com_readObject(
    connection *connection,
    gxObject* object,
    unsigned char attributeOrdinal)
{
    int ret;
    message data;
    gxReplyData reply;
    mes_init(&data);
    reply_init(&reply);
    if ((ret = cl_read(&connection->settings, object, attributeOrdinal, &data)) != 0
        (ret = com_readDataBlock(connection, &data, &reply)) != 0 ||
        (ret = cl_updateValue(&connection->settings, object, attributeOrdinal, &repl
    {
    }
    mes_clear(&data);
    reply_clear(&reply);
    return ret;
}
```

Writing values

Writing values to the meter is very simple. You just Update Object's propery and then write it. In this example we want to update clock time of the meter.

Note!
Data type must be correct or meter returns usually error. If you are reading byte value you can't write UIn16.

Java | C# | Delphi | ANSI C++ | ANSI C | Python

```
void writeObject(GXDLMSObject item, int attributeIndex) throws Exception
{
    GXReplyData reply = new GXReplyData();
    byte[] data = Client.write(item, attributeIndex);
    readDataBlock(data, reply);
}
```

Disconnecting

Last you must close the connection by sending disconnecting request. If you do not close connection correctly your next connection attempt will fail.

Java | C# | Delphi | ANSI C++ | ANSI C | Python

?

```
void close() throws Exception
{
    if (Media != null)
    {
        GXReplyData reply = new GXReplyData();
        readDLMSPacket(Client.disconnectRequest(), reply);
        Media.close();
    }
}
```

Keep Alive

If you do not read anything from the meter you must send a keep alive message. Best way is make a connection, read all data from the meter and close connection.

Java | C# | Delphi | ANSI C++ | ANSI C | Python

?

```
GXReplyData reply = new GXReplyData();
readDLMSPacket(Client.keepAlive(), reply);
```

Image updating

Updating new firmware to the meter is easy.

Java | C# | Delphi | ANSI C++

?

```
GXReplyData reply = new GXReplyData();
//Check that image transfer ia enabled.
readDataBlock(Client, Media, Client.read(target, 5), reply);
Client.updateValue(target, 5, reply.getValue());
if (!target.getImageTransferEnabled())
{
    throw new Exception("Image transfer is not enabled");
}

//Step 1: Read image block size.
readDataBlock(Client, Media, Client.read(target, 2), reply);
Client.updateValue(target, 2, reply.getValue());

// Step 2: Initiate the Image transfer process.
readDataBlock(Client, Media, target.imageTransferInitiate(Client, Identification, da

// Step 3: Transfers ImageBlocks.
int[] imageBlockCount = new int[1];
readDataBlock(Client, Media, target.imageBlockTransfer(Client, data, imageBlockCount

//Step 4: Check the completeness of the Image.
readDataBlock(Client, Media, Client.read(target, 3), reply);

Client.updateValue(target, 3, reply.getValue());

// Step 5: The Image is verified;
readDataBlock(Client, Media, target.imageVerify(Client), reply);

```

```
// Step 6: Before activation, the Image is checked;
//Get list to imaages to activate.
readDataBlock(Client, Media, Client.read(target, 7), reply);
Client.updateValue(target, 7, reply);
boolean bFound = false;
for (GXDLMSImageActivateInfo it : target.getImageActivateInfo())
{
    if (it.getIdentification().equals(Identification))
    {
        bFound = true;
        break;
    }
}

//Read image transfer status.
readDataBlock(Client, Media, Client.read(target, 6), reply);
Client.updateValue(target, 6, reply);
if (target.getImageTransferStatus() != ImageTransferStatus.IMAGE_VERIFICATION_SUCCES
{
    throw new RuntimeException("Image transfer status is " + target.getImageTransfer
}
if (!bFound)
{
    throw new RuntimeException("Image not found.");
}
//Step 7: Activate image.
readDataBlock(Client, Media, target.imageActivate(Client), reply);
```

Sending and receiving data

Data is not always fit to one packet. In this case data can be split to blocks and one block to frames. Gurux DLMS component handles this automatically.

When a full frame is received check if there is more data available using IsMoreData method. If there is more data, generate a new read message using ReceiverReady method and check are there all data received again.

After receiving the full frame check that there wasn't any errors from err.

Reading Frame is implemented as follows:

Java | C# | Delphi | ANSI C++ | ANSI C | Python

```
public void readDLMSPacket(byte[] data, GXReplyData reply)
        throws Exception {
    if (data == null || data.length == 0) {
        return;
    }
    Object eop = (byte) 0x7E;
    // In network connection terminator is not used.
    if (dlms.getInterfaceType() == InterfaceType.WRAPPER
            && Media instanceof GXNet) {
        eop = null;
    }
    Integer pos = 0;
    boolean succeeded = false;
    ReceiveParameters<byte[]> p =
            new ReceiveParameters<byte[]>(byte[].class);
    p.setAllData(true);
    p.setEop(eop);
    p.setCount(5);
    p.setWaitTime(WaitTime);
    synchronized (Media.getSynchronous()) {
        while (!succeeded) {
            writeTrace("<- " + now() + "\t" + GXCommon.toHex(data));
            Media.send(data, null);
            if (p.getEop() == null) {
                p.setCount(1);
            }
            succeeded = Media.receive(p);
            if (!succeeded) {
                // Try to read again...
                if (pos++ != 3) {
                    System.out.println("Data send failed. Try to resend "
                            + pos.toString() + "/3");
                    continue;
                }
                throw new RuntimeException("Failed to receive reply from the device
            }
        }
        // Loop until whole DLMS packet is received.
        while (!dlms.getData(p.getReply(), reply)) {
            if (p.getEop() == null) {
                p.setCount(1);
            }
            if (!Media.receive(p)) {
                throw new Exception("Failed to receive reply from the device in give
            }
        }
    }
    writeTrace("-> " + now() + "\t" + GXCommon.toHex(p.getReply()));
    if (reply.getError() != 0) {
        throw new GXDLMSException(reply.getError());
    }
}
```

Reading Block is implemeted as follows:

Java C# Delphi ANSI C++ ANSI C Python

```
/**
    * Reads next data block.
    *
    * @param data
    * @return
    * @throws Exception
    */
void readDataBlock(byte[] data, GXReplyData reply) throws Exception {
    if (data.length != 0) {
        readDLMSPacket(data, reply);
        while (reply.isMoreData()) {
            data = dlms.receiverReady(reply.getMoreData());
            readDLMSPacket(data, reply);
        }
    }
}
```

Authentication using challenge

When authentication is High or above secret is used. After connection is made client must send challenge to the server and server must accept this challenge. This is done checking is Is Authentication Required after AARE message is parsed. If authentication is required client sends challenge to the server and if everything succeeded server returns own challenge that client checks.

Java C# Delphi ANSI C++ ANSI C Python

```
//Parse reply.
Client.parseAAREResponse(reply);
//Get challenge Is HLS authentication is used.
if (Client.getIsAuthenticationRequired())
{
    reply = readDLMSPacket(Client.getApplicationAssociationRequest());
    Client.parseApplicationAssociationResponse(reply);
}
```

Transport security

DLMS supports tree different transport security . When transport security is used each packet is secured using GMAC security. Security level are:

- Authentication
- Encryption
- AuthenticationEncryption

Using secured messages (glo services) is easy. Before security can be used following properties must set:

- Security
- SystemTitle
- AuthenticationKey
- BlockCipherKey
- FrameCounter

Java C# Delphi ANSI C++ ANSI C Python

```
GXDLMSSecureClient sc = new GXDLMSSecureClient();
sc.getCiphering().setSecurity(Security.ENCRYPTION);
//Default security when using Gurux test server.
sc.getCiphering().setSystemTitle("GRX12345".GetBytes("ASCII");
```

?

Thats it. If you have any questions or ideas how to improve Gurux.DLMS component let us know or ask question in a Forum