Manual for the creation of modules for OpenSCADA

Contents table

Introduction

This manual is made to help in building the modules for OpenSCADA. The module creation may be required if you wish to add the support for new data source or other extension to the OpenSCADA. Since OpenSCADA is extremely modular system, all interfaces of interaction with the external environment are implemented by expanding it with specialized modules of following types:

- "Data Bases"
- "Communication Interfaces, Transports"
- "Protocols of the communication interfaces"
- "Data Sources and Data Acquisition"
- "Archives(Histories) (of messages and values)"
- "User Interfaces (GUI, TUI, WebGUI, speach, signal ...)"
- "Additional modules"

To create the OpenSCADA modules you need to have an experience in C/C++ programming language, build system AutoTools, as well as basic knowledge of Linux and the distribution you are using.

1. Creating a New Module

Modules in OpenSCADA are the shared libraries, which are dynamically connected to the OpenSCADA core at the time of the program execution. Many of the modules can be disabled, connected and updated during the execution from modules scheduler. Modules can also be included into the OpenSCADA core during building, by an argument **-enable-{ModName}=incl** to the "configure" configuration script, what can you learn from building manual. OpenSCADA modules can be of seven types according to the present modular subsystems. For now the modules for the OpenSCADA are written in the "**C++**" programming language, although in the future the bindings to other languages may appear.

In order to facilitate the creation of new modules in the sources tree, the branches of each subsystem provide the "**=Tmpl=**" directory with the module's template for the appropriate subsystem. The developer of the new module can copy this directory with the name of the new module. You can create modules in the OpenSCADA sources tree or as an independent project of the external module for the OpenSCADA.

1.1. Creation of the module in the sources tree of the OpenSCADA project

It makes sense to create new modules in the sources tree of OpenSCADA project in case of further plans for the transfer of a new module to the OpenSCADA. Since the module should not be contrary to the spirit of open source project and to the license, on the basis of which the OpenSCADA is developed and distributed; the license of the new module obviously should be one of the free licenses.

The procedure of a new module creation based on the template with its inclusion to the sources tree is easier than such procedure for the external module and includes the following steps:

1. Getting the sources tree of the OpenSCADA project.
   For a working branch:
   ```
   $ svn co svn://oscada.org/trunk/OpenSCADA
   ```
   For a stable release branch (not desirable because only corrections are accepted for the stable LTS releases):
   ```
   $ svn co svn://oscada.org/tags/openscada_0.8.0
   ```
2. Copy the template directory with the "NewMod" name of the new module (for example, for the "DB" subsystem):
   ```
   $ cd OpenSCADA/src/moduls/bd
   $ cp -r =Tmpl= NewMod; cd NewMod
   $ rm -rf .svn po/.svn
   ```
3. Editing the "module.cpp" file.
   Change the names of the functions of a module's including with the name of the new module:
   "**TModule::SAt bd_Tmpl_module(int n_mod)**" — bd_NewMod_module
   "**TModule *bd_Tmpl_attach(const TModule::SAt &AtMod, const string &source)**" — bd_NewMod_attach
   Information about the module in the "module.cpp" file, namely the area:
   ```
   //*************************************************
   //* Modul info!                                   *
   #define MOD_ID          "NewMod"
   #define MOD_NAME        _("DB NewMod")
   #define MOD_TYPE        SDB_ID
   #define VER_TYPE        SDB_VER
   #define MOD_VER         "0.0.1"
   #define AUTHORS         _("MyName MyFamily")
   #define DESCRIPTION     _("BD NewMod description.")
   #define MOD_LICENSE     "GPL2"
   ```
4. Edit the module's building configuration in the "Makefile.am" file:
   ```
   EXTRA_DIST = *.h po/*

   oscd_modul_LTLIBRARIES = db_NewMod.la
   ```

```
db_NewMod_la_CXXFLAGS = $(NewMod_CFLAGS)
db_NewMod_la_LDFLAGS = -module -avoid-version -no-undefined $
(NewMod_LDLAGS)
db_NewMod_la_SOURCES = module.cpp
db_NewMod_la_LIBTOOLFLAGS = --tag=disable-static

if NewModIncl
db_NewMod_la_CXXFLAGS += -DMOD_INCL
install:
endif

I18N_mod = $(oscd_modulpref)NewMod
```

5. Adding an entry of the new module to the end of the subsystem's section (for us it is "> DB modules") of the OpenSCADA building system configuration file (OpenSCADA/configure.in):

```
AX_MOD_EN(NewMod,[disable or enable[=incl] build module
DB.NewMod],disable,incl,
[
    AC_MSG_RESULT(Build module: DB.NewMod)
    AC_CONFIG_FILES(src/moduls/bd/NewMod/Makefile)
    DBSub_mod="${DBSub_mod}NewMod "
    #>> Modules checkings
    # The code checks the external libraries of the module
    if test $enable_NewMod = incl; then
        LIB_CORE="${LIB_CORE} moduls/bd/NewMod/.libs/*.o "
        ModsIncl="${ModsIncl}bd_NewMod "
    fi
])
```

6. Now, the new module can be built in the OpenSCADA after the reorganization of the building system:

```
$ autoreconf -if
$ configure --enable-NewMod
$ make
```

7. Publication. The formation of the patch with your module and send it to the OpenSCADA developers:

```
$ cd OpenSCADA; make distclean; rm -f src/moduls/bd/NewMod/Makefile.in
$ svn add src/moduls/bd/NewMod
$ svn diff > NewMod.patch
```

1.2. Creation of an external module for OpenSCADA

Creation of an external module for OpenSCADA may make sense in the case of the development the integration interface with business (commercial) systems that require proprietary interaction code, as well as in the case of other commercial interfaces implementations, in which the module for the OpenSCADA acquire the status of the separate project, that is distributed and maintained independently, often in the form of binary buildings for a specific platform and version of OpenSCADA. The license of such modules, respectively, can be arbitrary.

The procedure for creation of a new external module based on the template is largely similar to the previous procedure and includes the following steps:

1. Getting the sources of the OpenSCADA project. For an external module as a source of template you can use any OpenSCADA source files of version more than 0.8.0 because it is necessary to copy only the "=Tmpl=" directory and several files for build.
2. Copy the template directory with the "NewMod" name of the new module (for example, for the "DB" subsystem). Creating and copying the necessary files for an external module. Further the information files of the project "COPYING", "NEWS", "README", "AUTHORS" and "ChangeLog" must be filled according to the nature of the new module.
```
$ cp -r OpenSCADA/src/moduls/bd/=Tmpl= NewMod
$ rm -rf NewMod/.svn NewMod/po/.svn
$ touch NewMod/{NEWS,README,AUTHORS,ChangeLog}
$ cp OpenSCADA/I18N.mk NewMod/
```
3. Editing the "module.cpp" file, similar to the appropriate item in the previous section.
4. Edit the module's building configuration in the "Makefile.am" file, similar to the appropriate item in the previous section.
5. Editing the "configure.in" configuration file of the build system:
   "**AC_INIT([DB.Tmpl],[0.0.1],[my@email.org])**" — information about the module: name, version and e-mail of the project.
   "**AM_CONDITIONAL([TmplIncl],[test])**" — **AM_CONDITIONAL([NewModIncl], [test])**
6. Installing the OpenSCADA development package. Because the module is an external one and the OpenSCADA source files are needed only at the first stage of the module's creation, you must install the OpenSCADA development package (openscada-devel), which contains the header files and libraries.
7. Now you can build the new module, after formation of the building system
```
$ autoreconf -if
$ configure
$ make
```

2. Module's API

OpenSCADA API for the developer of OpenSCADA and modules for it is exhaustively, in a formal form, described in the paper OpenSCADA API, which should always be on hand for the development of OpenSCADA. This document focuses on the detailed explanation of the main points of the modular API.

The inheritance of root object-class of the module from the *TModule* class via a class of modular subsystem is common for all modules, which means that there is a common part of the module's interface, it'll be discussed below. In general, to present the architecture of modules in the context of the overall OpenSCADA architecture, it is strongly recommended to have before eyes the overall OpenSCADA class diagram!

The entry point of any module are the following functions:

- *TModule::SAt module(int n_mod), TModule::SAt bd_DBF_module(int n_mod)* — used to scan the list and information about all modules in the library. The first function is used to implement the modules in the external shared library, and the second when linking them to the OpenSCADA core.
- *TModule *attach(const TModule::SAt &AtMod, const string &source), TModule *bd_Tmpl_attach(const TModule::SAt &AtMod, const string &source)* — used for direct connection-opening the selected module by creating the root object of the module, inherited from the *TModule*. The first function is used to implement the modules in the external shared library, and the second when linking them to the OpenSCADA core.

In the constructor of the root module's object, inherited from the *TModule*, it is necessary to define the common meta information of the module:

- *mId* — module's ID is passed in the constructor's argument;
- *mName* — the name of the module;
- *mDescr* — the description of the module;
- *mType* — the type of the module;
- *mVers* — the version of the module;
- *mAutor* — the author of the module;
- *mLicense* — the distribution license of the module;
- *mSource* — the source/origin of the module, typically it is the full path to the shared library file with the code of the module.

As well as to initiate the environment of the module with the following functions:

- *void modFuncReg(ExpFunc *func);* — Registration of the module's exporting function. This function is the part of the intermodule interaction mechanism, which registers an internal function of the module for an external call by the function's name and its pointer relative to the module's object.

For the convenience of a direct addressing to the root object of the module from any module's object below in the hierarchy it is recommended to determine the global variable "mod" in the namespace of the module with its initialization in the module's root object constructor. Also, for the transparent translation of the module's text messages it is recommended to define the template of the function for call messages' translation of the module "**_({Message})**":

```
#undef _
#define _(mess) mod->I18N(mess)
```

For the purpose of general module's management the *TModule* class provides a number of virtual functions that can be defined in the root object of the module with the implementation of the necessary response to the OpenSCADA core commands to the module:

- *void load_( );* — Loading the module. It is called on the stage of loading the module's configuration from the configuration file or database.

- *void save_( );* — Saving the module. It is called on the stage of the module's configuration saving to the configuration file or database, usually initiated by the user.
- *void modStart( );* — Starting the module. It is called on the execution stage of background module's functions tasks, if any are provided by the module.
- *void modStop( );* — Stop the module. It is called on the stopping the execution stage of background module's functions tasks, if any are provided by the module.
- *void modInfo(vector<string> &list);* — Request the list of the information properties of the module. This function of the *TModule* class provides the standard set of properties of the module ("Module", "Name", "Type", "Source", "Version", "Author", "Description", "License"), which can be extended by additional properties of this module.
- *string modInfo(const string &name);* — Request for the specified information item. The processing of requests for additional properties of this module is made.
- *void postEnable(int flag);* — Connecting the module to the dynamic tree of object. It is called after the starting module.
- *void perSYSCall(unsigned int cnt);* — Call from the system thread at regular intervals for 10 seconds and the seconds counter <*cnt*>. It can be used to execute periodic, rare, service procedures.

All interface modules' objects inherit the *TCntrNode* node class, which provides the control interface mechanism, one of whose functions is to provide the configuration interface of the object in any OpenSCADA configurator. To solve the new module's tasks it may be necessary to expand the parameters of the configuration; it is made in the *void cntrCmdProc(XMLNode *opt);* virtual function. The contents of this function that adds a property in the simplest case has the following form:

```
void MBD::cntrCmdProc( XMLNode *opt )
{
    //> Get page info
    if(opt->name() == "info")
    {
        TBD::cntrCmdProc(opt);
        ctrMkNode("comm",opt,-1,"/prm/st/end_tr",_("Close opened
transaction"),RWRWRW,"root",SDB_ID);
        return;
    }
    //> Process command to page
    string a_path = opt->attr("path");
    if(a_path == "/prm/st/end_tr" &&
ctrChkNode(opt,"set",RWRWRW,"root",SDB_ID,SEC_WR)) transCommit();
    else TBD::cntrCmdProc(opt);
}
```

The first half of this function serves the "info" information requests with the list and properties of the configuration fields. The second half serves all the other commands on the getting, setting the value, and others. The *TBD::cntrCmdProc(opt);* call is used to obtain the inherited interface. More details on the appointment of the used functions are in the control interface, as well as in the source code of existing modules.

In addition to the control interface functions *TCntrNode* object provides standardized control mechanisms for the modification of the object's configuration, for the loading and saving the configuration to the storage. To complete the setting of the modification flag of object's data you can use the *modif()* and *modifG()* functions, and module specific actions for the loading and saving can be placed in the virtual functions:

- *void load_( );* — Loading an object from the repository.
- *void save_( );* — Saving object in the repository.

Typically the work with configuration is made through the *TConfig* object, which contains a set of specified properties. For a direct reflection of the module's object properties it is inherited from *TConfig*, and new properties are added by the following command:

```
fldAdd(new TFld("PRM_BD",_("Parameters cache table"),TFld::String,TFld::NoFlag,"30",""));
```

Loading and saving of the properties specified in the *TConfig* object, to/from the storage is made with the following command:

```
SYS->db().at().dataGet(fullDB(),owner().nodePath()+"DAQ",*this);
SYS->db().at().dataSet(fullDB(),owner().nodePath()+"DAQ",*this);
```

Where:

- *fullDB()* — the full name of the database-storage in the form: "**{DBMod}.{DBName}.{Table}**";
- *owner().nodePath()+"DAQ"* — total path to the node of the object-representative of the table in the configuration file
- **this* — this object, inherited from the *TConfig*.

2.1. "Data Bases (DB)" subsystem's module

This module is designed for the OpenSCADA integration with the DBMS, implemented by the module.

OpenSCADA interface to process the requests to the DB is presented by the objects and virtual functions of the calls from the OpenSCADA core:

- *TTipBD->TModule* — The root module's object of the "DB" subsystem:
  - *TBD *openBD(const string &id);* — It is called when you open or create by this module a new database object with <*id*> identifier.
- *TBD* — The database object:
  - *void enable( );* — Enabling of the database.
  - *void disable( );* — Disabling of the database.
  - *void load_( );* — Loading a database from a shared configuration storage.
  - *void save_( );* — Saving the database in a shared configuration storage.
  - *void allowList(vector<string> &list);* — Request the tables' list <*list*> in the database.
  - *void sqlReq(const string &req, vector< vector<string> > *tbl = NULL, char intoTrans = EVAL_BOOL);* — Handling of the <*req*> SQL-query to the database and receiving the results in the <*tbl*> table, if the selection request and the pointer are nonzero. When you set <*intoTrans*> to "true" a transaction must be opened for the request; to "false" - must be closed. This function should be implemented for a databases that support SQL-queries.
  - *void transCloseCheck( );* — The periodically called function to check the transactions and closing the old or contain many requests ones.
  - *TTable *openTable(const string &table, bool create);* — It is called when you open or create a new table's object.
- *TTable* — Table's object in the database:
  - *void fieldStruct(TConfig &cfg);* — Getting the current structure of the table in the *TConfig* object.
  - *bool fieldSeek(int row, TConfig &cfg);* — Sequential scan of table entries using the <*row*> exhaustive search and "false" return at the end with the addressing by the active, *keyUse()*, key fields.
  - *void fieldGet(TConfig &cfg);* — Request of the specified in the *TConfig* object record with the addressing by key fields.
  - *void fieldSet(TConfig &cfg);* — Transfer of the specified in the *TConfig* object record with the addressing by key fields.
  - *void fieldDel(TConfig &cfg);* — Deleting of the specified in the *TConfig* object record with the addressing by key fields.

2.2. "Transports" subsystem's module

The module of this type should provide OpenSCADA communications through the interface, often it is the network one, implemented by the module.

Software OpenSCADA interface to service incoming and outgoing requests through a network interface is presented by the objects and virtual functions of the calls from the OpenSCADA core:

- *TTipTransport->TModule* — The root module's object of the "Transports" subsystem:
  - *TTransportIn *In(const string &name, const string &db);* — It is called when you open or create by this module a new incoming transport object <*name*> with the <*db*> storage.
  - *TTransportOut *Out(const string &name, const string &db);* — It is called when you open or create by this module a new outgoing transport object <*name*> with the <*db*> storage.
- *TTransportIn* — The transport's object of the processing the incoming requests, the function of the server. Incoming requests received by the module through the implementation of a network interface must be sent to the specified in the configuration incoming protocol *protocol()* via the *mess()* function:
  - *string getStatus( );* — Call to get the specific status of the interface.
  - *void setAddr(const string &addr);* — Setting the address of transport. Can be overridden for the processing and verification of module-specific address format of the transport.
  - *void start();* — Start of the transport. When you start the incoming transport the task, that waits for requests from the outside, is typically created.
  - *void stop();* — Stop of the transport.
- *TTransportOut* — The transport's object of the processing the outgoing requests, the function of the client:
  - *string getStatus( );* — Call to get the specific status of the interface.
  - *void setAddr(const string &addr);* — Setting the address of transport. Can be overridden for the processing and verification of module-specific address format of the transport.
  - *void start( );* — Start of the transport. When you start the outgoing transport the actual connection to the remote station is established for the interfaces that works by the connection. At this time, the errors can occur if the connection is impossible, and the transport should return to the stopped state.
  - *void stop( );* — Stop of the transport.
  - *int messIO(const char *obuf, int len_ob, char *ibuf = NULL, int len_ib = 0, int time = 0, bool noRes = false);* — Processing the requests from the OpenSCADA core to send data over the transport. The <*time*> waiting time of connection is specified in milliseconds, having a nonzero value it must replace the same transport's timeout in its general configuration. <*noRes*> is used by the protocols for the exclusive blocking the transport for the time of being working with it and for its own blocking exclusion by the function. The package to be sent is specified in the <*obuf*> buffer with the <*len_ob*> length, and the buffer and its size for the response are specified in the <*ibuf*> and <*len_ib*>. The outgoing buffer <*obuf*> may be empty (NULL) if you want to check for further response or responses, received without a request, the mode of broadcasting. If the response buffer is not specified (NULL), the response waiting will not be realized.

2.3. "Transport protocols" subsystem's module

The module of this type should provide OpenSCADA with the protocol layer communications, implemented by the module, for data access of the external systems and for OpenSCADA data from external systems.

Software OpenSCADA interface to implement the protocol layer is presented by the objects and virtual functions of the calls from the OpenSCADA core:

- *TProtocol->TModule* — The root module's object of the "Protocols" subsystem:
  - *void itemListIn(vector<string> &ls, const string &curIt = "");* — The list of sub-elements of the incoming protocol, if the protocol provides them. It is used when selecting an object in the configuration of incoming transport.
  - *void outMess(XMLNode &io, TTransportOut &tro);* — The transfer of data by the objects of the OpenSCADA core in the <*in*> XML tree to the remote system via the <*tro*> transport and the current outgoing protocol. Presentation of data in the <*in*> XML tree is non-standardized and specific to the logical structure of the protocol. This data are serialized (converted into a sequence of bytes according to the protocol) and are sent via the specified <*tro*> outgoing transport by the *messIO()* function above.
  - *TProtocolIn *in_open(const string &name)* — It is called when you open or create by this module the new <*name*> incoming transport protocol object.
- *TProtocolIn* — Protocol's object of the incoming requests processing from the incoming transport object *TTransportIn* above. For each session of the incoming request the object of the associated incoming protocol is created, which remains alive until the completion of a full "request->answer" session. Address of the transport, which opened an instance of the protocol, is specified in the *srcTr()*:
  - *bool mess(const string &request, string &answer, const string &sender);* — Transfer of the <*request*> data sequence to the protocol's object for their parsing according to the protocol's implementation, with the specification of the requesting object's address in the <*sender*>. This protocol's function should process the request, generate the response in <*answer*> and return "false" in the case of the completeness of the request. If the request is not complete, it is necessary to return "true" to indicate the "expectation of the completion" for the transport, the first part of the request should be saved in the context of the protocol's object.

2.4. "Data acquisition" (DAQ) subsystem's module

The module of this type should provide the real-time data acquisition from the external systems or their formation in the calculators, implemented by the module.

The software OpenSCADA interface to implement access to real-time data is presented by the objects and virtual functions of the calls from the OpenSCADA core:

- *TTipDAQ->TModule* — The root module's object of the "Data acquisition" subsystem:
  - *void compileFuncLangs(vector<string> &ls);* — Request in the <*ls*> the list of user programming languages supported by the module.
  - *void compileFuncSynthHighl(const string &lang, XMLNode &shgl);* — Request the syntax rules <*shgl*> of the specified user programming language <*lang*>.
  - *string compileFunc(const string &lang, TFunction &fnc_cfg, const string &prog_text);* — Calling the compilation of user's procedure <*prog_text*> and creation an object of the function's execution based on the <*fnc_cfg*> for the specified user programming language <*lang*> of this module. Returns address of the compiled function's object, ready for execution.
  - *bool redntAllow( );* — A flag of support the redundancy mechanisms by module. Should be overridden and return "true" if supported, otherwise "false".
  - *TController *ContrAttach(const string &name, const string &daq_db);* — It is called when you open or create a new controller's object <*name*> by this module in the <*db*> storage.

- *TController* — Data source controller's object. In the context of the object is usually runs the task of the periodic or scheduled polling of real-time data of one physical controller or a dedicated physical data block. In the case of data getting by the packages, they are placed directly into the archive associated with the parameter's attribute *TVAl::arch()*, and the current value is set by the *TVAl::set()* function with the attribute "sys"=true:
  - *string getStatus( );* — Call to get the specific status of the controller.
  - *void enable_( );* — The enabling of the controller. Usually at this stage the initialization of the parameters' objects and their interfaces in the form of attributes is made, the attributes can sometimes be requested from the associated remote source.
  - *void disable_( );* — Disabling the controller.
  - *void start_( );* — Start of the controller. Usually at this stage the task of periodic or scheduled polling is created and started.
  - *void stop_( );* — Stop of the controller.
  - *void redntDataUpdate(bool firstArchiveSync = false);* — The operation of obtaining data from the backup station. It is called automatically by the redundancy scheme's service task and before the start to synchronize archives with the parameter <*firstArchiveSync*>.
  - *TParamContr *ParamAttach(const string &name, int type);* — It is called when you open or create a new parameter's object <*name*> with the <*type*> type.
- *TParamContr->TValue* — The object of the data source controller's parameter. It contains the attributes with real data in a set, defined by the physically accessible data. The values and attributes are taken from the task of the controller's polling in the asynchronous mode, or requested at the time of query in the synchronous mode with the help of methods of this object's inherited type *TValue*:
  - *void enable( );* — To enable the parameter. The formation of the attributes set and filling them with the value of unreliability is made.
  - *void disable( );* — To disable the parameter.
  - *void setType(const string &tpId);* — It is called to change the <*tpId*> type of the parameter and can be processed in the module's object to change its own data.
  - *TVal* vlNew( );* — It is called at the stage of a new attribute creation. It can be overridden to implement a particular behavior within its own, inherited from the *TVal*, class during the access of the attribute.
  - *void vlSet(TVal &val, const TVariant &pvl);* — Called for an attribute with a direct recording mode *TVal::DirWrite* (synchronous or write to an internal buffer of the object) when set to record the values in the immediate physical controller or buffer object.
  - *void vlGet(TVal &val);* — It is called for an attribute with a direct reading mode *TVal::DirRead* (synchronous mode, or reading from an internal buffer of the object) when reading the value in order to directly read the value from the physical controller or object's buffer.
  - *void vlArchMake(TVal &val);* — It is called at the stage of creation the values archive with the <*val*> attribute as the source in order to initialize the qualitative characteristics of the archive's buffer according to the characteristics of the data source and polling.

2.5. "Archives" subsystem's module

This type of the module is used for archiving and maintaining the history of OpenSCADA messages and real-time data obtained in the "Data acquisition" subsystem with the help of the module.

The OpenSCADA software interface to implement an access to the archived data is presented by the objects and virtual functions of the calls from the OpenSCADA core:

- *TTipArchivator->TModule* — The root module's object of the "Archives" subsystem:
  - *TMArchivator *AMess(const string &id, const string &db);* — It is called when you open or create a new message archiver <*id*> by this module in the <*db*> storage.
  - *TVArchivator *AVal(const string &id, const string &db);* — It is called when you open or create a new values archiver <*id*> by this module in the <*db*> storage.
- *TMArchivator* — The messages archiver object with its own archiving method and storage location:
  - *void start( );* — Start of the archiver. The archiver starts to receive messages and place them into storage.
  - *void stop( );* — Stop of the archiver.
  - *time_t begin( );* — Data beginning in the archiver accordingly with the current state of the storage.
  - *time_t end( );* — Data end in the archiver in accordance with the current state of the storage.
  - *void put(vector<TMess::SRec> &mess);* — Call to place the <*mess*> messages in the storage.
  - *void get(time_t b_tm, time_t e_tm, vector<TMess::SRec> &mess, const string &category = "", char level = 0, time_t upTo = 0);* — Request of the <*mess*> messages in the archive for the <*b_tm*>...<*e_tm*> interval according to the category template <*category*> and the level, with the restriction on the request time to <*upTo*>.
- *TVArchivator* — The values archiver object with its own archiving method and storage location:
  - *void setValPeriod(double per);* — It is called when the archivers values frequency is changed.
  - *void setArchPeriod(int per);* — It is called when the archiving frequency is changed.
  - *void start( );* — Start of the archiver. The archiver starts to receive messages and place them into storage.
  - *void stop(bool full_del = false);* — Stop of the archiver with the ability to completely remove its data from the storage, if the <*full_del*> is set.
  - *TVArchEl *getArchEl(TVArchive &arch);* — Request the object-representative of the <*arch*> archive, served by the archiver.
- *TVArchEl* — The representative object of the values in the archiver's storage:
  - *void fullErase( );* — It is called to complete remove a part of the archive in the archiver.
  - *int64_t end( );* — End time of the archive in the archiver.
  - *int64_t begin( );* — Start time of the archive in the archiver.
  - *TVariant getValProc(int64_t *tm, bool up_ord);* — The request for the processing of one value from the archive for the time <*tm*> and fine-tuning to the upper value in the sampling grid <*up_ord*>.
  - *void getValsProc(TValBuf &buf, int64_t beg, int64_t end);* — The request for the getting values group processing <*buf*> for the specified period of time.
  - *void setValsProc(TValBuf &buf, int64_t beg, int64_t end);* — The request for the setting values group processing <*buf*> for the specified period of time.

2.6. "User interfaces" (UI) subsystem's module

The module of this type should provide a user interface by its own method. The root object of the module is *TUI->TModule*, which does not contain specific interfaces, and the user interface is formed in accordance with the implemented concept and mechanisms, for example, of the graphic primitives library.

2.7. "Specials" subsystem's module

The module of this type should implement specific functions that are not included in any of the above subsystems. The root object of this module is *TSpecial->TModule*, which does not contain specific interfaces and specific functions are formed according to their needs, using all the features of OpenSCADA API.