**CLI Language Tutorial**

|  |
| --- |
|  |

Table Of Contents

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
| 02 | 2006-11-23 | Weblib | R. Verstraete | F. Denissen |
| 01 | 2006-02-08 | Weblib | R. Verstraete | F. Denissen |
| **ED** | **DATE** | **CHANGE NOTE** | **APPRAISAL AUTHORITY** | **ORIGINATOR** |
| CLI Language : Tutorial | | | | |

List Of Figures 4

List Of Tables 4

History 5

Scope 7

Referenced Documents 8

Related Documents 8

List of Abbreviations 9

Glossary 10

Assumptions 11

1 Introduction 12

1.1 Creating commands 12

1.2 Typical errors when designing CLI commands 12

1.2.1 Layering 12

1.2.2 Specialisation 13

1.2.3 Non-existing instances 13

1.2.4 Single command to create different objects 13

1.2.5 Multiple resource identifiers point to the same instance 14

1.2.6 No or not enough res-id for show command 14

1.2.7 Not all res-id on leaf node for show command 15

1.2.8 Too many sections used for show command 15

1.2.9 Output of info configure is not replayable 15

1.3 General guidelines 15

2 CLI LANGUAGE 17

2.1 Naming conventions 17

2.2 Commands 17

2.2.1 Types 17

2.2.2 Access rights 17

2.2.3 Options and parameters 18

2.3 Nodes 18

2.3.1 Node definition 18

2.3.2 Resources 19

2.3.3 Navigation 19

2.3.4 Node creation 20

2.3.5 Node deletion 20

2.3.6 Resource attributes 20

2.4 Parameters 20

2.4.1 Parameter types 21

2.4.2 Basic types 22

2.5 Command location 22

3 XML 24

3.1 Syntax rules 24

3.2 Validation 24

3.3 Reading XML Schemas 25

4 The Execution Engine 27

4.1 Basic concepts 27

4.2 Fields 27

4.3 Phases 28

4.4 Access rights 30

4.4.1 Values 30

4.4.2 Assigning access rights 31

4.5 Loading of fields in the context 31

4.6 How fields are used by the agent 32

4.6.1 Reading 32

4.6.2 Writing 34

4.6.3 Searching 34

5 patterns FOR BEGINNERS 36

5.1 Type patterns 36

5.1.1 Builtin type pattern 36

5.1.2 Enumeration pattern 42

5.1.3 Boolean pattern 45

5.1.4 Combined parameters 46

5.2 Node patterns 49

5.2.1 Scalar config node 49

5.2.2 Static config node 51

5.2.3 Dynamic config node 61

5.2.4 Intermediate node 65

5.2.5 Admin node 65

5.2.6 Show node 68

6 pATTERNS FOR EXPERIENCED CLI DESIGNERS 78

6.1 Type patterns 78

6.1.1 Bit masks 78

6.1.2 Ignore SNMP errors 87

6.1.3 Index combines values of different fields 88

6.1.4 Object values 90

6.2 Node patterns 94

6.2.1 Port lists 94

6.2.2 Value lists 98

6.2.3 Name as res-id instead of index 102

6.2.4 Name as parameter instead of index 105

6.2.5 Fast ranges 114

7 Patterns for expert cli designers 116

7.1 Validation of strings typed by the operator 116

8 Index handling 117

8.1 Introduction 117

8.2 Nomenclature 117

8.3 Implementation 118

8.3.1 C++ classes 118

8.3.2 XML files 125

9 backwards compatibility 128

9.1 Applicability 128

9.2 Avoiding backward compatibility problems 128

9.3 Solutions in case of incompatibilities 128

9.3.1 Obsolete parameters 128

9.3.2 Obsolete alternatives 130

9.3.3 Obsolete commands 131

9.4 Addition of explicit type-identifier 134

9.4.1 When to use ? 134

9.4.2 Basic implementation 134

10 Build 136

10.1 Build products 136

10.2 MIBCO 136

11 Host TESTING 137

11.1 Help file 137

11.2 Simulated environment 137

List Of Figures

Figure 1. commands 17

Figure 2. nodes 19

Figure 3. parameters 21

Figure 4. Phases for **in** 28

Figure 5. Phases for **create** 28

Figure 6. Phases for **modify** 29

Figure 7. Phases for **delete** 29

Figure 8. Phases for **display** 29

Figure 9. Phases for **poll** and **sample** if the node is dynamic and the instance exists 29

Figure 10. Phases for **poll** and **sample** if the instance does not exist or if the node is static 29

List Of Tables

Table 1. The default access rights of a field 31

History

**Ed. 01**

Created on February 22, 2005 by Frank Denissen

Proposal 01: March 21, 2005 by Frank Denissen

Changes: Initial version, with only patterns for beginners.

Proposal 02 : March 23, 2005 by Frank Denissen

Changes:

* Added information on typical command design errors.
* Added patterns for more experienced CLI designers.

Proposal 03 : April 13, 2005 by Frank Denissen

Changes:

* Added information on more typical command design errors.
* Added information on the sample command.

Proposal 04 : September 1, 2005 by Frank Denissen

* Corrected the man page title specifications.

Proposal 05 : September 2, 2005 by Frank Denissen

* Add an example to explain why mixing mib keys and columns in one parameters introduces problems.

Proposal 06 : September 22, 2005 by Frank Denissen

* Adapt document front page
* Explain short name concept
* Replace “This node” by “This command”

Proposal 07 : January 12, 2006 by Frank Denissen

* Describe new handling of indices
* Add references to important documents
* Some small corrections
* Describe XML for direct access to the database via datadriven proxies

Release on 2006-02-08.

**Ed. 02**

Proposal 01 : February 14, 2006 by Frank Denissen

* Add chapter on backward compatibility.
* Add section on how the CLI Agent uses the type procedures.
* Add section on how to enforce detailed format for show commands.
* Some small editorial changes.

Proposal 02 : March31, 2006 by Frank Denissen

* Add section on how to implement fast ranges
* Add section on invisible prefix

Proposal 03: April 14, 2006 by Frank Denissen

* Correct unclear description

Proposal 04: September 28, 2006 by Frank Denissen

* Add information on the simulated context
* Complete the section “Typical errors when designing CLI commands”

Proposal 05: November 23, 2006 by Frank Denissen

* New features for R3.3 :
  + Obsolete alternatives
  + Obsolete commands
  + Hidden ranges
  + Manual splitting of table headers

All/non-editorial changes with respect to Ed. 02 Proposal 04 are indicated with revision bars; new text is underlined and removed text is indicated with strikethroughs.

Scope

This document is a tutorial for development of CLI using XML.

Referenced Documents

1. CLI agent use case sheets

3HH 00404 AAAA DSZZA

1. CLI agent language definition

3HH 00405 AAAA DSZZA

1. TMN/CLI TRS

3HH 00055 AAAA DTZZA

1. Data Driven ConfigData

3HH-00904-ABAA-DTZZA

/vobs/dsl/sw/flat/ConfigDataXml/

1. ISAM CLI XML user manual tutorial

3HH-00014-3027-DFZZA

Related Documents

1. Error descriptions in XML

3HH-00518-AAAA-DSZZA

List of Abbreviations

CLI Command Line Interface

MIB Management Information Base

PDU Protocol Data Unit

SNMP Simple Network Management Protocol

XML eXtensible Mark-up Language

Glossary

Operator A person that gets information on the actual behaviour of the system or changes the configuration of the system accessing the system via one of the management interfaces (telnet, craft terminal)

Assumptions

1. Introduction
   1. Creating commands

This document is intended as tutorial for developing CLI for iSAM.

Before you start coding the commands, you must define them in words. Try to place yourself in the place of the operator and answer following questions :

* What does the operator want to do with the new feature ?
* How does he want to use it ?
* What (logical) actions must he do to make the new feature active ?
* What must he do to enable this new feature for a new board/customer, ...
* What must he do to disable this new feature for a board/customer ...
* What actions must he do to deactivate the feature ?
* What are the interactions with other features ?
* How does he verify that the feature is properly working ?
* What is the minimum number of things he has to do (simplest case) ?

Where you can find the information you need:

* Read the feature description
* Study the related MIB
* Read the PB document.
* Look how other vendors have implemented it (for example: login in to a Timetra box and look for similar commands)
* Talk to the designers of the feature

The next steps are :

* Rewrite the commands to be conformant with our CLI syntax. For this you need to know the structure of our CLI language. This is explained in chapter 2.
* Specify the commands in detail in XML. The XML syntax is explained in chapter 3. How to write the commands in XML is explained starting from chapter 5.

Chapter 4 describes briefly how the execution engine of the CLI works. This is necessary to understand what you have to specify in the XML.

Every section builds further on the previous section. So it is very important you read the first time everything in sequence ! Beginners should read chapters 1 till 5 (included) and 8.

* 1. Typical errors when designing CLI commands
     1. Layering

Error :

* A command related to a lower layer object (for example : connection layer) is a subnode of a higher layer object (for example : network layer).

Solution :

* Preferred solution : the command related to higher layer objects are in the command tree put after the commands for lower layer objects.
* Other solution : the command related to a higher layer object is a subnode of the command related to the lower layer object.
  + 1. Specialisation

Example :

* All L3 filters have a number of common parameters (filtering on ip-address, network port , …)
* TCP L3 filters have an additional number of parameters (filtering on TCP port range, flags, …
* IGMP L3 filters have an additional, but different, number of parameters (filtering on message type, …)

Error : the specialisation is implemented in the wrong way.

* One basic command is created with all the common parameters
* A subnode with the specific parameters is created for each of the specialised types

Consequence : The operator can for each L3 filter object set the common parameters AND the TCP related parameters and the IGMP related parameters, while the purpose was to set the common parameters AND the TCP related parameters OR the IGMP related parameters.

Solution :

* Create a node for each of the specialised types + optionally one for a non-specialised object.
* Repeat the common parameters in each node.
* Provide in the CLI the necessary filters to assure that an info command of the non-specialised node does not show the instances of the specialised types.
  + 1. Non-existing instances

Error :

* A number of object types have a large number of res-id.
* A node is created with as res-id the ones that are common between all object types.
* A subnode is created for each of the object types with the non-common res-id.
* There exists no real-world object instance (MIB table, database table, ….) identified by the common res-id.

Consequence : the Execution Engine will not be able to display any elements when an info command is executed on the node.

Solution :

* Limit the number of res-id of the common node till a real-world instance exists.
* If no such real-world instance can be found : remove the common node and add all res-id to the subnodes.
  + 1. Single command to create different objects

Error : one single command is used to create multiple objects.

* This can be very easily recognised if one node needs to work with 2 or more RowStatus fields.

Consequence : the info, modify or delete command will fail if the creation of one of the two objects failed or if only one of the objects was created by an external manager.

Solution :

* Provide in the **configure** node a command for each of the objects. The commands are preferable placed next to each other in the command tree.
* If it is a customer requirement that a command exists that creates the different objects at the same time, create than an additional command to do so in the **admin** node.
  + 1. Multiple resource identifiers point to the same instance

Example :

* The index of the MIB table is an ip-address.
* One of the columns in the MIB table is the prefix-length
* In CLI is the res-id formatted as <ip-address>/<prefix-length>

Consequence : the operator will be confused because if he creates first the instance 123.123.123.123/12 and then instance 123.123.123.123/16 and he does info, he will find out that only 123.123.123.123/16 exists and that 123.123.123.123/12 magically disappeared.

Solution : the prefix-length must be a separate parameter.

The root cause of this problem is that one parameters contains both an identifier and an attribute. With following example, I will try to explain why the CLI engine interpretes the things in the wrong way.

Extended example :

Suppose you have two girls. Each girl is identified by its name (Daphne or Anita) and has one attribute: the color of her dress (red or blue).

We make now a CLI command to manage a girl and we define that a girl is identified (RES-ID) by its name AND the color of her dress (!!!WRONG!!!) and that it has no attributes (PARAMETER)

So for CLI there are 4 possible girls (and not 2 like in the real-world) :

* Daphne with the red dress
* Daphne with the blue dress
* Anita with the red dress
* Anita with the blue dress

You notice now also that in the real world Daphne can change her dress from red to blue, while in CLI this is not possible. Changing the dress would require from the CLI in one command it deletes the girl "Daphne with the red dress" and creates the girl "Daphne with the blue dress". CLI only supports CREATE, MODIFY, DELETE (and in this case even no MODIFY because for CLI a girl has no attribute).

The problem becomes worse if CLI uses another system (for example SNMP) to verify if a particular girl exist in the real-world and when this system is lazy. So if the CLI asks to this system 'Does "Anita with the red dress" exist ?' and the system verifies only if Anita exists and not if Anita wears a red dress then this system will answer yes on this question, even if Anita wears now a blue dress. You can understand that the CLI will sometimes   
gives the wrong information to its user.

Note :

You can enforce that the SNMP becomes less lazy by setting the flag **isFilter\_c** in the access rights of the attribute fields. SNMP will then ALWAYS read the attribute and do the necessary filtering.

* + 1. No or not enough res-id for show command

One should define as many res-id for a show command so that every line of the output in table format is uniquely identified.

The table format will not be possible in case no res-id are defined. The display elements of all instances will be considered as scalar variables : they will be displayed all after each other in list format.

Display elements of different instances will be grouped together if not enough res-id are provided.

* + 1. Not all res-id on leaf node for show command

In principle must for show commands (not for configure and admin commands) all res-id be on the leaf node. In this way the operator has more possibilities : if a res-id is located in a higher node, then it is only possible to see instances for a particular value for this res-id, not for all values of this res-id.

* + 1. Too many sections used for show command

Sections should only be used in following conditions :

* We want to show some scalar variables that are common for all instances. In this case the definition of the section in XML contains **is-common=”true”**
* We have so many display-ele in detailed mode that it is very hard for an operator to have an overview. Example : we have 25 display-ele to be shown : 5 general ones, 10 related to the transmit side, 10 to the receive side. In this case we should use two sections : one for the transmit side and one for the receive side.

Note : don’t use sections if you only have 3 display-ele : you will only confuse the operator.

* + 1. Output of info configure is not replayable

It is mandatory that the output of **info configure flat** is replayable. Often this is not the case. Typical problems are:

* Commands are not added to the command tree in the proper sequence. Commands that must be executed first must be put higher in the command tree.
* Passwords and secrets are replaced by “**\*\*\***” or replaced by an empty string. Use an alternative field instead which allows to enter a plain text password or an encrypted password (see the password attribute of configure system security operator). Make sure that the password is printed in encrypted format.
* The read-out parameter value is not equal to the entered parameter value (example: activate when setting, active when read)
  1. General guidelines

Reuse as much as possible : do not create basic types if they already exist. Before creating a new basic type always **grep** first in **/vobs/dsl/source08/CLI\_Language/basicTypesMerged.xml** first to find similar basic types.

Follow strictly the guidelines the man pages :

* Add only text that you understand yourself.
* Don’t add screenshots.
* Don’t repeat information which is already generated (names of parameters, help on parameters and display elements).
* All words in the title of a man page must start with a capital.
* The title must end respectively with **Configuration Command**, **Status Command** or **Management Command** for subnodes of respectively the **configure**, **show** and **admin** top-nodes.

Try to use the same names for parameters, values, … as in similar commands.

1. CLI LANGUAGE
   1. Naming conventions

All keywords an operator can enter (command names, node names, parameter names, enumeration values, ….) must be lower-case and must start with an alphabetic character (‘**a**’…’**z**’). Individual words within the keyword are separated from each other by a single dash ‘**-**‘.

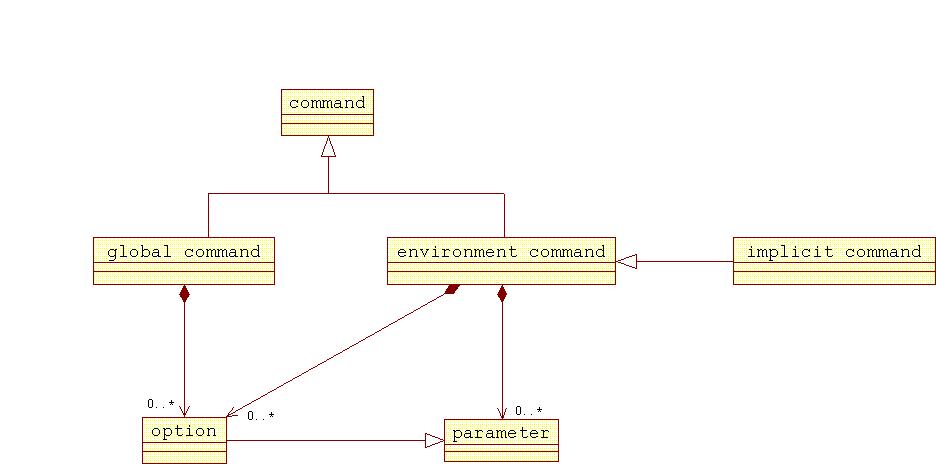
* 1. Commands
     1. Types

All commands of the CLI language can be divided in two groups : global and environment commands.

Global commands (like **logout**) can be entered in any context and have the same effect in each context.

Environment commands (like **info**) can only be entered in some contexts (for example : **info** can only be entered in the **configure** node) and have a different effect in each context : the purpose of the command (for example : showing the configuration) is the same, but the implementation or the generated output is different.

Some commands are also implicit(含蓄的). For example: when you enter only the node name **configure system security snmp community fden** you run an implicit command that changes your context to the specified node and that even creates this node if it did not exist before.



1. commands
   * 1. Access rights

Not every operator can execute every command. Access to commands is granted via the operator profile. It is possible that an operator can execute a given command in one context and not in another context.

Which operator profile gives access to which command is formally described in XML.

* + 1. Options and parameters

Commands can have options and parameters (see 2.4).

Options are special parameters that are context-insensitive. For example: you can enter the option **detail** after **info** in any context.

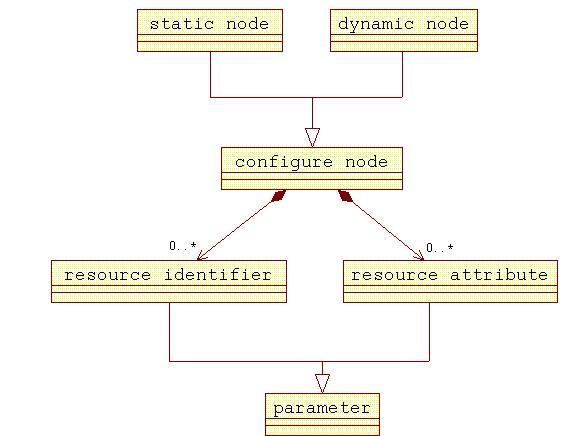
* 1. Nodes
     1. Node definition

A command definition tree, further abbreviated to “command tree” is a structure of nested command nodes from which CLI commands can be derived. A command node consists of a node name and zero or more resource identifiers. The resource identifiers behave like parameters (see 2.4), but identify a particular resource (see 2.3.2). For example: **“public”** in **configure system security snmp community “public”** is an unnamed resource identifier of the node **community**.

One such command node identifies one context. A CLI command can be derived from a command tree starting from the root node, but a command with the same meaning and impact can also be derived from a lower level node or sub-node. Examples of equivalent commands:

* **info configure system security snmp community “public”** in any node
* **info system security snmp community “public”** in node **configure**
* **info security snmp community “public”** in node **configure system**
* **info snmp community “public”** in node **configure system security**
* **info community “public”**in node **configure system security snmp**
* **info** in node **configure system security snmp community “public”**

The root node is at the highest level. Directly below the root node are important command nodes like **configure** and **show**.



1. nodes
   * 1. Resources

There is a one-to-one mapping between configurable resources and sub-nodes of the **configure** node. Example: one can manipulate(操作) the configuration of the SNMP community string with the name **public** by changing the resource attributes of the node **configure system security snmp community “public”** or its sub-nodes (if present).

* + 1. Navigation

One can navigate to a node by entering its name, starting from the root node or relative to the current node. Example: we can navigate to **configure system security snmp community “public”** via:

* **configure system security snmp community “public”** in any node
* **system security snmp community “public”** in node **configure**
* **security snmp community “public”** in node **configure system**
* **snmp community “public”** in node **configure system security**
* **community “public”**in node **configure system security snmp**

Following additional commands are provide to navigate in the command tree:

* **exit all** moves to the root node.
* **exit** moves to the parent node.
* **back** moves to the previous node.

Not every operator can navigate in any node. This navigation will be restricted by the access rights of the operator profile (see 2.2.2).

There are also nodes to which no operator can navigate. For example: one can not navigate to the **leaf-nodes** of **show** because the implicit command linked to this node shows run-time information on this node.

* + 1. Node creation

A sub-node of the **configure** node that corresponds to a configurable resource that an operator can create, is called a dynamic node.

An operator can create such a dynamic node by navigating into it. The system will then automatically create the node. The possibility to create nodes will of course be limited by the access rights of the user.

It is also possible that the system creates additional sub-nodes in other nodes (for example in the **show** node) due to the creation of a new dynamic node in the **configure** node.

* + 1. Node deletion

A dynamic node can be deleted by placing **no** in front of the node name. Example: **configure system security snmp no community “public”** deletes the specified node and all its sub-nodes. The possibility to delete nodes will of course be limited by the access rights of the operator.

* + 1. Resource attributes

The value of resource attributes can be changed by entering the name of the resource attribute followed by the new value. Example: **password plain:”secret”** sets the value of the resource attribute with name password to the plain text string **“secret”.**

Resource attributes can be set to their default value by entering **no** followed by the name of the resource attribute. Example: **no password** sets the value of the resource attribute with name **password** to the default value (no password required).

Changing a resource attribute to a default value via *parameter-name* *default-value* has exactly the same effect as the comand **no** *parameter-name*.

* 1. Parameters

Parameters are characterised by a name and a type (see 2.4.1).

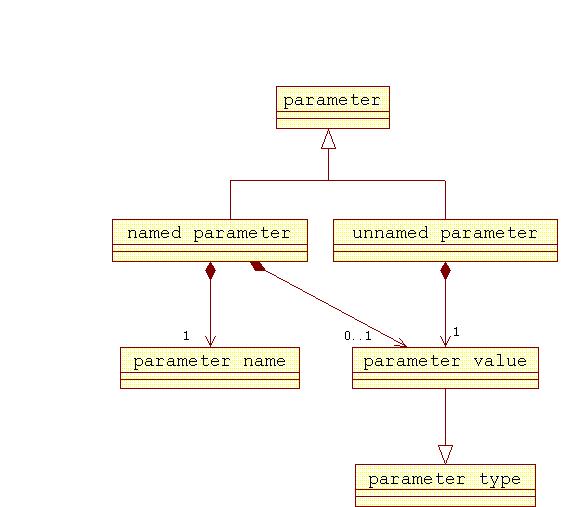
Parameters can be named or unnamed.

Named parameters are entered as 2 tokens : a keyword and a value (example: **password plain:”secret”**). Some commands (like **info**) require that only the keyword is entered for a parameter.

Only the value token is entered for unnamed parameters (example: **“public”** in the node name **community “public”**).

The designer can not freely choose which parameters are named or unnamed. Following restrictions exists :

* The first res-id must be unnamed
* Only boolean parameters and parameters that trigger a command in the **admin** node may be unnamed
* Options may be unnamed
* All other parameters must be named.



1. parameters
   * 1. Parameter types

Parameter values belong to a parameter type. Parameter types limit the parameter values to strings that the system can understand.

Parameter types consist of 1 to N fields with separators inbetween. Each field belongs to a basic type (see 2.4.2).The separator is in most cases the colon ‘:’.

Parameter types are defined via XML. The name of parameter types always starts with an uppercase character and can contain, optionally a domain name. The domain name is separated from the parameter name by “**::**”. The first character of **each** word in a parameter type name is written in uppercase, the remaining characters in lowercase. All words are just concatenated.

The sequence in which the fields appear can differ in case one of the fields belongs to an enumeration type. As example, we give for the parameter type **ExtendedFileName** some possible values (the first field is an enumeration field with as possible values **file** and **ftp,** *host-name*,*user-name*,*password* and *local-file* are also fields) :

**file:***local-file-name*

**ftp:***host-name***@***user-name***:***password***/***local-file-name*

are built from combinations of basic types.

Variable length native types:

integers

strings (representing operator chosen names or descriptions)

object identifiers

binary strings

…

Fixed native types

ip-address

fixed-length strings

enumeration types

…

Structured types: build from a combination of native types

* + 1. Basic types

The value of basic types can typically be assigned to 1 SNMP variable or to 1 database cell.

Basic types can be divided in two groups : fixed length basic types and variable length basic types.

Examples of variable length basic types are :

integers

printable strings (representing operator chosen names or descriptions)

SNMP object identifiers

binary strings

The length of a variable length basic type is in most cases limited

The definition of logical length depends on the basic type: number of characters for printable strings, number of bytes for binary strings, number of numbers in the object identifier for SNMP object identifiers, …

Examples of fixed length basic types are :

ip-address

fixed-length printable strings

fixed-length binary strings

enumeration types (limited list of allowed strings)

…

Basic types are defined in XML and follow the same naming conventions as parameter types. The name spaces of basic and parameter types are completely separated: so it is possible that there are parameter types that have the same name as a basic type.

* 1. Command location

Commands that change the configuration (typically read-write objects in the MIB) are placed in the **configure** top node. The operator should be able to execute an **info** command in the **configure** node, write the output of the command to a file and use this as a script to configure a system in its initial state (with default database).

Commands that trigger one-shot actions (clear statistics, copy a file, download a file, start a test, …) must be placed in the **admin** node.

The leaf nodes in the **show** top-node show the actual status of the system (typically read-only objects in the MIB). It is allowed to show also configuration parameters in a **show** command if this is usefull of an operator: for example to compare the configured with the actual state. Do not make an additional **show** command for configuration parameters only.

1. XML
   1. Syntax rules

XML is the abbreviation of eXtensible Markup Language. We use this notation to describe the CLI commands.

What is the minimum that you need to know ?

That there are elements also called tags and attributes and attribute values.

* elements can contain content
* attributes have values.

Example: <ELEMENT NAME="attribute">HELLO!</ELEMENT>

This are the rules for well-formed XML documents :

|  |  |
| --- | --- |
| 1. The XML declaration must begin the document. | <?xml version="1.0" encoding="UTF-8"?> |
| 2. Elements that contain data must have both start and ending tags | <commands>        <command handler-type="info" class="CommandDefinition"/>  </commands> |
| 3. Elements that do not contain data and use only a single tag must end with /> | <commands>        <command handler-type="info" class="CommandDefinition"/>  </commands> |
| 5. Elements may nest but not overlap | <default-command>    <command handler-type="in" class="CommandDefinition">     </default-command>   </command> |
| 6. Attribute values must be quoted | <default-command>          <command handler-type="in" class="CommandDefinition"/>   </default-command> |
| 7. The only escape sequences used are &amp; = ‘&‘ &lt; = ‘<’ &gt; = ‘> &quot; = ‘”’ | <alternatives>    <alternative identifier="altr-e" value="&quot;ALTR-E&quot;"/>   </alternatives> |
| 8. Comment | <!- - This is comment - - > |

* 1. Validation

For validating the XML a XML-schema is used. On validation following checks are made :

|  |  |
| --- | --- |
| The order of the TAGS or elements. | Example : if the order is invalid, you will get following error message :  [Error] configNode.xml:81:13: The content of element type "node" must match "(help,man-page,default-command,commands?,subnode\*,parameters?,fields?,node\*)". |
| Invalid name used for element or attribute | Example : if element name help1 is used instead of help, you will get following error message : [Error] configNode.xml:19:14: Element type "help1" must be declared.  [Error] configNode.xml:80:13: The content of element type "node" must match "(help,man-page,default-command,commands?,subnode\*,parameters?,fields?,node\*)". |
| Escape sequence not used | Example : escape sequence &quot; not used for ‘"’ :  source : <command handler-type="in" class=""CommandDefinition""/>  [Fatal Error] configNode.xml:12:42: Element type "command" must be followed by either attribute specifications, ">" or "/>". |
| The value of an attribute is not allowed | Example : [Error] configNode.xml:18:53: Datatype error: Value 'errorerrorerror' must be one of [all, admin, none].  [Error] configNode.xml:80:13: The content of element type "node" must match "(help,man-page,default-command,commands?,subnode\*,parameters?,fields?,node\*)". |

You find the XML schemas in the directory **/cm4/tools/BELL/xml/xmlCommon** :  basicTypes.xsd, parameterTypes.xsd, node.xsd, ...

You can validate an XML file with following command : /cm4/tools/BELL/xml/xmlCommon/validatexml <file name>

Example :

/cm4/tools/BELL/xml/xmlCommon/validatexml /vobs/dsl/source08/CLI\_Agent/TEST/topNode.xml

* 1. Reading XML Schemas

|  |  |
| --- | --- |
| <xsd:element name="basic-types">     <xsd:complexType>       <xsd:sequence>         <xsd:element name="basic-type" type="BasicType"  minOccurs="1" maxOccurs="unbounded"/>          </xsd:sequence>       </xsd:complexType>  </xsd:element> | Declares a new element or tag with the name basic-types which contain an other element basic-type.  The element basic-type is of the type BasicType and is mandatory (minOccurs="1") and there is no limit to the number of elements (maxOccurs=”unbounded”). If the element is not mandatory then minOccurs must be "0". |
| <xsd:complexType name="BasicType">          <xsd:sequence>              <xsd:element name="help" type="xsd:string"  minOccurs="1" maxOccurs="1" />              <xsd:element name="length" type="LengthType"  minOccurs="0" maxOccurs="1" />             <xsd:element name="option" type="xsd:string"  minOccurs="0" maxOccurs="1" />              <xsd:element name="restrictions"  type="RestrictionListType"  minOccurs="0" maxOccurs="1" />          </xsd:sequence>          <xsd:attribute name="name" type="xsd:string" use="required" />          <xsd:attribute name="class" type="xsd:string" use="required" />         <xsd:attribute name="option" type="xsd:string" use="optional" />     </xsd:complexType> | Elements of the type BasicType must have in SEQUENCE the tags :  help : mandatory  length : optional  option : optional  restrictions : optional  Elements of the type BasicType must have following attributes : name : mandatory class : mandatory option : optional |

1. The Execution Engine
   1. Basic concepts

The Execution Engine is responsible for the execution of the commands specified by the operator.

The Execution Engine has for each CLI session a context. This context is its working environment.

The Execution Engine works only with fields. These fields are specified in the parameter-type, node and handler XML files (see later).

The Execution Engine will, for each command entered by the operator, load the relevant fields in the context. It can be that, while the execution progresses, additional fields are loaded in the context, while others are removed.

* 1. Fields

Each field loaded in the context can store following values as strings :

* entered value : the value in the format as entered by the operator
* common value : the entered value, but converted to common representation
* current value : the original value of a MIB variable, but converted to common representation
* command value : the value as shown to the operator

Apart from the stored values are there two more calculated values that are often used:

* modified value : the common value if this exists, the current value otherwise
* read value : the read value if this exists, the common value otherwise

Each field has a type. This type is defined in the basic-type XML files. This type restricts the values that can be given to a field. Each type is linked to a C++ type class. This C++ type class contains procedures to :

* validate the operator input as he types in the characters
* expand the operator input if requested
* convert the value as entered by the operator to common value
* write the common value in a SNMP PDU
* read a value from a SNMP PDU and assign it to current value
* convert common or current value to command value
* …. and many more

The format of the common and current values is determined by the basic class:

* printed with format "%ld" for SignedIntegerType
  + printed with format "%lu" for UnsignedIntegerType
  + the string itself for PrintableStringType
  + a lower-case hexadecimal string with **:** as separator for **BinaryStringType** (example: **07:ef:3b**)
  + unsigned number separated by dots for **IpAddressType** and **ObjectType** (example : **128.37.125.44**)

Entered and command value are also strings. The format is as expected by the operator. It can differ from the format of common and current values. Example: an operator will enter **127.0.0.1** for an **HexIpAddressType**, but the corresponding common and current values will be **7f:00:00:01** because the base class is **BinaryStringType** and not **IpAddressType**.

The type classes have following conversion routines :

* **convertCommandToCommonRepresentation()** : converts the entered value to common value. These procedures copy by default the entered value to the common value.
* **getCommandValue()** and **getShortCommandValue()**: convert the common or current value to command value. **getShortCommandValue()** is only used during show to have a more compact representation. These procedures copy by default the common or current value to the display value.
* **convertCommonToMibRepresentation()** : takes the common value and puts in the SNMP PDU. This function may only be defined in the base type classes.
* **convertMibToCommonRepresentation()** : reads the SNMP PDU and assigns the value to current value. This function may only be defined in the base type classes.
  1. Phases

Each command is executed by the Execution Engine in a number of phases. These phases are called : **in**, **create**, **prepare**, **modify**, **delete** and **display**.

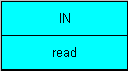
The phases **create**, **prepare**, **modify** and **delete** have each two sub-phases : **read** and **write**. **In** and **display** have only one sub-phase : **read**.

The XML designer can specify what must be done with a field for each phase (see 4.4). Before discussing this, we will give first an overview of all phases.

**In** is used when :

* A command is entered with only node names and keys
* The commands **exit** or **back** are used
* The default command is **in**

Note : **in** is executed for each node entered by the operator as he types in the node names and resource identifiers.



1. Phases for **in**

**Create** is used when at least one parameter is entered and :

* The default command of the current node is **action**
* The default command of the current node is **configure** and the node is dynamic and the instance does not yet exist (verified in the **in** phase)



1. Phases for **create**

**Modify** is used when at least one parameter is entered and the default command is **configure** and :

* The node is static
* The node is dynamic and the instance does exist (verified in the **in** phase)



1. Phases for **modify**

**Delete** is used when the node name is preceeded by **no**.



1. Phases for **delete**

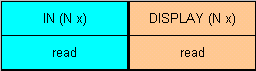
**Display** is used for nodes with default command **show** or for the **info** command.

The **in** phase is for **display** executed for :

* The node entered by the operator
* For each dynamic sub-node without res-id

The **display** phase is executed for :

* The node entered by the operator
* Each instance of each sub-node



1. Phases for **display**

**Poll** is used for each node with default command **poll**.

The **display** phase is repeated at regular time intervals as specified by the operator.



1. Phases for **poll** and **sample** if the node is dynamic and the instance exists



1. Phases for **poll** and **sample** if the instance does not exist or if the node is static

**Sample** is used for each node with default command **sample**.

The **display** phase occurs only once.

* 1. Access rights
     1. Values

The CLI designer must specify what must be done with each field in each sub-phase. He does this by changing the access right of a field.

Changing the access rights is the only way the CLI designer has to influence the dynamic behaviour of a command (when is which SNMP message sent). All the other XML definitions describe the static behaviour.

Specifying the correct access rights is one of the skills that is required to become an experienced CLI designer.

For each phase he can specify :

* **mayBeRead**
* **mayBeWritten**
* **mayBeCalculated**
* **mayBeUsed**

**mayBeRead** means that the Execution Engine will try to give the field a value by reading the MIB variable via SNMP or - for data not accessible via SNMP – from its storage via the node class. It does not mean that the Execution Engine will read the MIB. It will only do so if all necessary conditions are fullfilled :

* the field does not have a value yet
* a MIB variable is linked to the field
* the MIB variable is not read before (with or without success)
* all the fields that are used as indices of the MIB table have a value

**mayBeWritten** means that the Execution Engine will try to write its value to the MIB variable via SNMP or – for data not accessible via SNMP – to its storage via the node class. It does not mean that the Execution Engine will write the MIB. It will only do so if all necessary conditions are fullfilled :

* the field has a value
* a MIB variable is linked to the field
* the MIB variable is not written before (with or without success)
* all the fields that are used as indices of the MIB table have a value

**mayBeCalculated** means that the Execution Engine must call the **calculateValue()** function of the C++ type class related to the field to give the field a value. This function has no effect for most C++ type classes. It is only used for some very specialised classes.

**mayBeUsed** means that the Execution Engine will not do anything with the field. It is just left in the context and keeps its values and its status. It can be that other fields will use its value or give it a value. Fields that are only used as MIB index will have this access right.

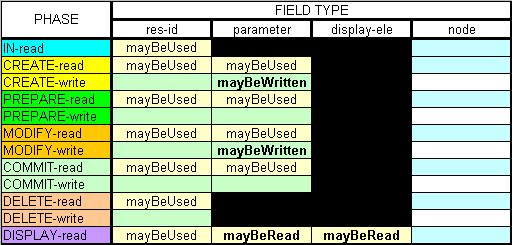
The CLI designer can in the access rights also specify modifiers that change the way that the Execution Engine handles the field. These modifiers are :

* **isFilter**
* **isReadableViaNode**
* **isWritableViaNode**

**isFilter** means that the Execution Engine must read the field via SNMP or via the node class, even if this field has already a value. The node instance will be considered to be invalid in case the read value is not equal to the value the field had before.

**IsReadableViaNode/ isWritableViaNode** means that the Execution Engine must read/write the field via the node class and not via SNMP. This will be the case for data not accessible via SNMP.

* + 1. Assigning access rights



The default access rights of a field

The CLI designer must specify what must be done with each field in each phase. He does this by selecting the appropriate access rights for each field and specifying this via the **access** attribute in the field specification. The default values are given in Table 1. Note that the default values depend on the type of field.

The values that can be given as access right are restricted. Neither the XML code generator nor the Execution Engine will complain if invalid combinations are made (for example : specifying both **mayBeCalculated** and **mayBeRead** for a phase or specifying **mayBeWritten** for a display-ele).

For the light yellow sub-phases following values may be given : **mayBeUsed**, **mayBeRead**, **mayBeCalculated**.

For the light blue sub-phases the same values may be given or no value.

For the light green and white sub-phases the value **mayBeWritten** or no value may be given.

For black sub-phases no value may be assigned.

* 1. Loading of fields in the context

The (visible and auxiliary) fields related to res-id of the last node entered by the operator till the top-node are always in the context. The CLI Agent adds thus implicitely **mayBeUsedDuringIn\_c | mayBeUsedDuringPrepare\_c | mayBeUsedDuringCreate\_c |mayBeUsedDuringModify\_c | mayBeUsedDuringCommit\_c | mayBeUsedDuringDisplay\_c** to the access rights specified in XML.

Node related fields or parameter and display-ele fields of higher nodes are never loaded in

Node related fields are only loaded in the context for a given sub-phase in case the corresponding access attribute has a value for the read sub-phase or for the write sub-phase.

Parameter fields and display-ele (visible and auxiliary) fields of the last node entered by the operator are always loaded in the context, except for the **in** command. The CLI Agent adds thus implicitely **mayBeUsedDuringPrepare\_c | mayBeUsedDuringCreate\_c | mayBeUsedDuringModify\_c | mayBeUsedDuringCommit\_c | mayBeUsedDuringDisplay\_c** to the access rights specified in XML.

Node related fields or parameter and display-ele fields of higher nodes are never loaded in the context.

It is clear that the Execution Engine will not do anything with fields that are not loaded in the context.

The loading and unloading of the fields can be followed via the CLI traces. The start of each phase is also indicated.

* 1. How fields are used by the agent
     1. Reading

The CLI\_Agent will go over all active fields and do the following :

* if a value is entered by the operator : assign the value entered by the operator to entered value
* if no value is entered by the operator and the field has a default value: assign the default value to entered value

The CLI\_Agent will then repetitively go over all active fields and do the following :

* try to give each field a common value:
  + take the next field if the field already has a common value
  + call **convertCommandToCommonRepresentation()**
  + take the next field if the field has now a common value
  + call **calculateValue()** if the access attribute of this field indicates that this field must be calculated in this phase. This function can assign a common value to the field based on the value of other fields and/or the current activity. This function does by default nothing.
  + take the next field if the field has now a common value
* try to give each field a current value:
  + if the access attribute of this field indicates that this field must be read via the node and this was not done before: call the **getInstance()** function on the node class. This function must assign a current value to all relevant node fields.
  + if the field must be read and the field is linked to MIB column and all indices for this table have a modified value:
    - construct a GET PDU to read all active fields that belong to the same MIB table
    - call **convertCommonToMibRepresentation()** for all index fields
    - send the PDU and wait for the response
    - call **convertMibToCommonRepresentation()** for the read MIB objects to assign values to current value
    - call **distributeToOtherFields()** for each read MIB object to assign values to the current values for related fields. This is typically used for indices where you can calculate the value of identifier fields from the read index value of the index (see section 8)
  + take the next field

The loop stops if no progress is made: no more fields get a new current or common value.

Example :

* field A : the IP-address as entered by the operator. This field is the index of MIB table X.
* field B : the status of the IP-address as read from MIB table Y.
* field C : the interface-index as read from MIB table X. This field is the index of MIB table Y.

The actions taken by the CLI\_Agent while reading are :

Handling entered values

* field A : assign IP-address to entered value
* field B : nothing is done as nothing is entered by the operator and because there is no default value linked to the field.
* field C : nothing is done as nothing is entered by the operator and because there is no default value linked to the field.

Handling common values

* field A : call **convertCommandToCommonRepresentation()** : common value gets a value because there is an entered value
* take field B as field A has now a common value
* field B : call **convertCommandToCommonRepresentation()** but no common value is assigned because there is no entered value
* field B : **calculateValue()** is not called because this is not requested by the access attribute
* take field C
* field C : call **convertCommandToCommonRepresentation()** but no common value is assigned because there is no entered value
* field C : **calculateValue()** is not called because this is not requested by the access attribute

Handling current values

* field A : **getInstance()** of the node class is not called because this is not requested by the access attribute
* field A : no SNMP Get request is sent because no MIB column is associated to this field
* take field B
* field B : **getInstance()** of the node class is not called because this is not requested by the access attribute
* field B : as SNMP Get request is not sent because field B is a column from MIB table Y and field C that is an index of MIB table Y does not have a modified value
* take field C
* field C : **getInstance()** of the node class is not called because this is not requested by the access attribute
* field C : construct a GET PDU as field C is a column from a MIB table X and all indices (here only field A) has a common or current value
* field C : call **convertCommonToMibRepresentation()** for the common value of field A (the index)
* field C : send the PDU and wait for the response
* field C : call **convertMibToCommonRepresentation()** for field C : current value gets a value

Start the next loop as fields A and C got a value in the previous loop

Handling common values

* take field B as field A has a common value
* field B : same actions as shown before but nothing changes
* take field C
* field C : same actions as shown before but nothing changes

Handling current values

* field A : same actions as shown before but nothing changes
* field B : construct a GET PDU as field B is a column from a MIB table Y and all indices (here only field C) have a modified value
* field B : call **convertCommonToMibRepresentation()** for the modified value of field C
* field B : send the PDU and wait for the response
* field B : call **convertMibToCommonRepresentation()** for field B : current value gets a value
* field C : take field A as field C has a current value

Start the next loop as field C got a value in the previous loop

Handling common values

* field A : take field B as field A has a common value
* field B : some actions as shown before but nothing changes
* field C : same actions as shown before but nothing changes

Handling current values

* field A : same actions as shown before but nothing changes
* take field C as field B has a current value
* skip field C as it has a current value

Stop as no field got a new value

* + 1. Writing

The CLI\_Agent will first read as many fields as it can as explained in section 4.6.1.

The CLI\_Agent will then go over all active fields and do the following :

* take the next field if the field has no modified value or if the field is already written
* If the access attribute of this field indicates that this field must be written via the node and this:
  + call the **completeAction()** function of the node class. This function must store the value of the related fields and mark them as been written.
* If the field is linked to a MIB column and the access attribute of this field indicates that this field must be written:
  + construct a SET PDU for all active fields that must be written and that belong to the same MIB table and that have a modified value
  + call **convertCommonToMibRepresentation()** on all index fields to fill in the indices
  + call **convertCommonToMibRepresentation()** on all data fields to fill in the SNMP values
  + send the PDU and wait for the response
  + mark all fields as being written
* take the next field
  + 1. Searching

The CLI Agent will go into search mode during an **info** or **show** command in case there are still fields to be read while the normal read process described in section 4.6.1 has finished. This indicates that the CLI Agent must display multiple entries, typically because the operator did not enter all res-id.

This will be a recursive process. The CLI Agent will recursively switch from read to search mode and back until it has found all instances and all values of each instance.

In search mode there are two ways of progressing.

The first way is taken if the access attribute of a field that still must be read, indicates that the field must be read via the node and this was not done yet:

* The CLI Agent will call the **getNextContext()** function of the node to create a node-specific structure that contains information on which instances exist.
* It will call repetitively the **getNextInstance()** function of the node with the node-specific structure as a parameter. This function must assign values for the next instance to the related node fields
  + The CLI Agent stops repeating if the function returns **false** and continues with the next step.
  + The CLI Agent verifies if the entry is valid.
  + The CLI Agent skips this entry and calls **getNextInstance()** again if this entry is not valid.
  + The CLI Agent moves for all fields the current value to the common value so that during following search actions incompatible entries can be filtered out.
  + The CLI Agent restarts the read process as described in section 4.6.1. It is possible that it will go again in search mode for other fields.
* It will release the node-specific data structure

The second way is taken if the first is not applicable:

1. The CLI Agent sends a SNMP GETNEXT PDU to get the first instance for the MIB column linked to the field that still must be read.
2. The CLI\_Agent stops if a value is returned for another MIB column than the requested one.
3. The CLI Agent unpacks the indices and the returned value by calling the **convertMibToCommonRepresentation()** functions.
4. The CLI Agent verifies if the entry is valid.
5. The CLI\_Agent skips this entry and sends a GETNEXT PDU for the last returned value and restarts at step 2 if the entry is not valid.
6. The CLI Agent moves for all fields the current value to the common value so that during following search actions incompatible entries can be filtered out.
7. The CLI Agent restarts the read process as described in section 4.6.1. It is possible that it will go again in search mode for other fields.
8. The CLI\_Agent sends a GETNEXT PDU for the last returned value and restarts at step 2.

The CLI Agent considers an entry to be invalid when any of following conditions is true:

* The function **currentValueIsValid()** of the type class of one of the fields returns **false**. This is typical the case when the value does not match the restrictions defined in the basic-type.
* The current value of the field has not the same value (case sensitive !!!) as the common value. This is typically the case when a res-id entered by the operator does not match what is read.

1. patterns FOR BEGINNERS
   1. Type patterns
      1. Builtin type pattern
         1. When to use ?

Use this pattern for parameters where the operator must enter a built-in type (signed integer, unsigned integer, printable string, binary string, object-id, ip-address, …). Don’t use it in following cases :

* the value must be interpreted by the operator (for example: 1 = start, 2 = stop). Use in this case one of the patterns explained in sections 5.1.2, 5.1.3, 5.1.2, 6.1.1, 6.1.4, 6.2.1 and 6.2.2).
* it only makes sense for an operator if another parameter has a given value. For example : to specify how the system must gets its ip-address, it only makes sense for the operator to specify an ip-address in case he selects as method **manual** and not when he specifies **dhcp** (see section 5.1.4, 6.1.3)
  + - 1. Basic Construction : signed integer

In the MIB we find :

VlanId ::= TEXTUAL-CONVENTION

STATUS mandatory

DESCRIPTION

"A 12-bit VLAN ID used in the VLAN Tag header."

SYNTAX INTEGER (1..4094)

extendVlanMgntVlanId OBJECT-TYPE

SYNTAX VlanId

ACCESS read-write

STATUS mandatory

DESCRIPTION

"The VLAN-ID referring to the management VLAN.

The instance of this object only exists when extendVlanMgntMode

is set to in-band."

::= { plimSystemObject 5 }

In a basic-type XML file :

<basic-type name=”Vlan::Id” class=”SignedIntegerType”>

<help>the number identifying a vlan</help>

<restrictions>

<range min-included=”1” max-included=”4094” action=”accept” type=”signed”/>

</restrictions>

</basic-type>

In a parameter-type XML file :

<parameter-type name=”Vlan::Id”

<field name=”” basic-type=”Vlan::Id”/>

</parameter-type>

In a node XML file :

<parameter name=”mgnt-vlan” parameter-type=”Vlan::Id”>

<help>the number of the management vlan</help

<optional default=”4093L”/>

<fields>

<field name=””>

<**mib-var** tree-node=”NODE\_ extendVlanMgntVlanId”

table-name=”SnmpAgent::localScalarTable\_m”/>

</field>

</fields>

</parameter>

Properties :

* A new basic-type is created because we have encountered a new SNMP type.
* The class specified in the basic-type must be SignedIntegerType because the SNMP type is INTEGER, even if all allowed values are unsigned.
* The type of the range must be signed because the class of the basic type is SignedIntegerType.
* The ranges specified in the MIB are coded in the basic-type
* The name Vlan::Id of the basic-type and parameter-type is the same. This is possible because basic-types and parameter-types have distinct namespaces. It is recommended to use the same names wherever possible.
* The parameter is named.
* The name of the parameter is completely in lower case, words separated by dashes and less than 17 characters long. The name of a parameter without default value may be 22 characters long.
* The name of the fields in the parameter-type and the parameter are the same. The name is here an empty string, which is recommended in case a parameter has only one field.
* The default value is within the range specified in the basic-type. The operator can thus enter both **no mgnt-vlan** or **mgnt-vlan 4093** to set the default value for **mgnt-vlan**.
* The L after 4093 is necessary because all default values must have as type in C : **unsigned long**, **signed long** or **string**. The L specifies that we have a **signed long**. We must use this C-type because it is a SignedIntegerType. Note : (long)4093 would have the same effect.
* The default value is specified in the CLI, even if no default is specified in the MIB, because we know that this is the initial setting.
* The **mib-var** construct links the field to the MIB column.
* The reference used as **tree-node** value is generated by MIBCO
* The value of **table-name** is SnmpAgent::localScalarTable\_m because it is a scalar MIB variable on the ASAM. SnmpAgent::lanxScalarTable\_m must be used for a scalar MIB variable on the LANX.
* All help strings are in lower case ~~and maximum 56 characters long~~. Note that in practise there is no limit anymore, but we propose a new "soft limit": +- 3x56 chars.

Verification via CLI :

* Enter : **?**

**[no] mngt-vlan : the number of a vlan reserved for managing**

**default = 4093**

* Enter : **mngt-vlan ?**

**<Vlan::Id> ! the number identifying a vlan**

**range = [1…4094]**

Attention points :

* The help of the parameter and the default value are shown for **?**
* The name and help of the basic-type and the range are shown for **mgnt-vlan ?**
  + - 1. Variant : unsigned integer

Identical to signed integer with following differences :

* The class is UnsignedIntegerType. It must be used for the SNMP types GAUGE, COUNTER and TIMETICKS
* The type of the range is unsigned.
* The default value is written as 4093UL or (unsigned long)4093.
  + - 1. Variant : unsigned signed integer

Identical to unsigned integer with following differences :

* The class is UnsignedSignedIntegerType. It must be used when the SNMP type is INTEGER, but where the range contains only unsigned values and the maximum value can not be represented as a positive number in a **signed long**.
  + - 1. Variant : integer with read-only values

In a basic-type XML file :

<basic-type name=”Vlan::Id” class=”SignedIntegerType”>

<help>the number identifying a vlan</help>

<restrictions>

<range min-included=”0” max-included=”0” action=”accept” type=”signed”

can-be-entered=”false”/>

<range min-included=”1” max-included=”4094” action=”accept” type=”signed”/>

</restrictions>

</basic-type>

The operator can now enter all integers in the range 1 till 4094. He will not be able to enter 0, but the execution engine will accept that the value 0 is read (info/show) : the execution engine will not generate error messages or skip table entries.

* + - 1. Variant : integer with non-contiguous range

In a basic-type XML file :

<basic-type name=”Vlan::Id” class=”SignedIntegerType”>

<help>the number identifying a vlan</help>

<restrictions>

<range min-included=”1” max-included=”4094” action=”accept” type=”signed”/>

<range min-included=”4093” max-included=”4093” action=”refuse” type=”signed”/>

</restrictions>

</basic-type>

The operator will be able to enter all integers in the range from 1 till 4094, with the exception of 4093.

An alternative way of writing this is :

<basic-type name=”Vlan::Id” class=”SignedIntegerType”>

<help>the number identifying a vlan</help>

<restrictions>

<range min-included=”1” max-included=”4092” action=”accept” type=”signed”/>

<range min-included=”4094” max-included=”4094” action=”accept” type=”signed”/>

</restrictions>

</basic-type>

* + - 1. Variant : shifted integer

Sometimes it is necessary that the operator must type in a number, but that via SNMP the number plus an offset must be written.

This is for example the case for Shub::NetworkPort. The operator must enter a number from **0** to **7**, but **17** till **24** must be written via SNMP.

The basic type is defined as follows :

<basic-type name=”Shub::NetworkPort” class=”SignedIntegerType”>

<help>the network port number</help>

<help-range>[0…7]</help-range>

<restrictions>

<range min-included=”17” max-included=”24” action=”accept” type=”signed”/>

</restrictions>

<option>-17</option>

</basic-type>

Properties :

* The range 17 till 24 that is valid for SNMP is given as restriction.
* A special tag help-range must be specified which gives the range 0 till 7 that can be entered by the operator. This is necessary because the default range shown to the operator is based on the restrictions.
* A special tag option is added to inform the class SignedIntegerType that -17 (=7-24) must be added to the value entered by the operator. Options are additional parameters that are passed as last parameters to the constructor of the class to change its behaviour. The interpretation of the option is different for each class. For SignedIntegerType and UnsignedIntegerType is the first option the command value minus the common/current value. See the header file of the concerned class for more information.
  + - 1. Variant : printable string

In the MIB we find :

DisplayString ::= OCTET STRING

-- This data type is used to model textual information taken

-- from the NVT ASCII character set.

sysContact OBJECT-TYPE

SYNTAX DisplayString (SIZE (0..255))

ACCESS read-write

STATUS mandatory

DESCRIPTION

"The textual identification of the contact person

for this managed node, together with information

on how to contact this person."

::= { system 4 }

In a basic-type XML file :

<basic-type name=”Sys::ContactName” class=”PrintableStringType”>

<help>the name of the contact person</help>

<length min=”0” max=”255”/>

</basic-type>

In a parameter-type XML file :

<parameter-type name=”Sys::ContactName”

<field name=”” basic-type=”Sys::ContactName”/>

</parameter-type>

In a node XML file :

<parameter name=”contact” parameter-type=”Sys::ContactName”>

<optional default=”&quot;&quot;”/>

<fields>

<field name=””>

<**mib-var** tree-node=”NODE\_ sysContact”

table-name=”SnmpAgent::localScalarTable\_m”/>

</field>

</fields>

</parameter>

Properties :

* A new basic-type is created because this MIB variable has a length which is different from other MIB variables.
* The class specified in the basic-type must be PrintableStringType because the SNMP type is OCTET STRING and because the textual description of DisplayString promises that only printable ASCII characters will be used.
* The minimum and maximum length of the string is specified in the basic-type.
* The default value is written as &quot;&quot; because we want it to be the empty string “” in C.
  + - 1. Variant : printable string with limited character set

In a basic-type XML file :

<basic-type name=”Sys::RestrictedFileName” class=”PrintableStringType”>

<help>the name of a file or a subdirectory in a directory </help>

<length min=”0” max=”255”/>

<restrictions>

<range min-included=”’a’” max-included=”’z’” action=”accept” type=”char”/>

<range min-included=”’0’” max-included=”’9’” action=”accept” type=”char”/>

</restrictions>

</basic-type>

This construct allows only lower-case alfanumeric characters.

Following type allows all printable characters, with exception of a space, a slash ‘/’ and a dot ‘.’ :

<basic-type name=”Sys::FileName” class=”PrintableStringType”>

<help>the name of a file or a subdirectory in a directory </help>

<length min=”0” max=”255”/>

<restrictions>

<range min-included=”’!’” max-included=”’~’” action=”accept” type=”char”/>

<range min-included=”’/’” max-included=”’/’” action=”refuse” type=”char”/>

<range min-included=”’.’” max-included=”’.’” action=”refuse” type=”char”/>

</restrictions>

</basic-type>

Following construct allows the same character set as the previous one, but does additionally not allow the strings root and special :

<basic-type name=”Sys::FileName” class=”PrintableStringType”>

<help>the name of a file or a subdirectory in a directory </help>

<length min=”0” max=”255”/>

<restrictions>

<range min-included=”’!’” max-included=”’~’” action=”accept” type=”char”/>

<range min-included=”’/’” max-included=”’/’” action=”refuse” type=”char”/>

<range min-included=”’.’” max-included=”’.’” action=”refuse” type=”char”/>

<strings action=”refuse”>

<string name=”root”/>

<string name=”special”/>

</strings>

</restrictions>

</basic-type>

* + - 1. Variant : binary string

Identical to printable string with following differences :

* The class is BinaryStringType. It must be used for the SNMP type **OCTET STRING** in case it is not guaranteed that all bytes in the string are printable ASCII characters.
* The default value is written in hexadecimal format as &quot;00:f3:a7&quot;
  + - 1. Variant : object identifier

Identical to printable string with following differences :

* The class is ObjectType. It must be used for SNMP type **OBJECT IDENTIFIER**.
* The length element in the basic-type represents the minimum and maximum number of integers that can appear in the object identifier.
* The default value is written as &quot;4.3.6.127.2.14&quot;
  + - 1. Variant : symbolic object identifier

Identical to object identifier but the Execution Engine will try to replace the initial part of the object identifier by the best matching object name. Replace the class ObjectType by ObjectSymbolicType.

Warning : such basic types may only be used for show commands.

* + - 1. Variant : ip-address

In the MIB we find:

AsamSNTPServerIpAddress OBJECT-TYPE

SYNTAX IpAddress

ACCESS read-write

STATUS mandatory

DESCRIPTION

"IpAddress for the SNTP Server

of the SNMP agent."

::= {asamSNTPMIB 1}

In a basic-type XML file :

<basic-type name=”Ip::V4Address” class=”IpAddressType”>

<help>a IPv4 address</help>

</basic-type>

In a parameter-type XML file :

<parameter-type name=”Ip::V4Address”

<field name=”” basic-type=”Ip::V4Address”/>

</parameter-type>

In a node XML file :

<parameter name=”sntp-server” parameter-type=”Ip::V4Address”>

<help>the ip-address of the sntp server</help>

<optional default=”&quot;0.0.0.0&quot;”/>

<fields>

<field name=””>

<**mib-var** tree-node=”NODE\_ AsamSNTPServerIpAddress”

table-name=”SnmpAgent::localScalarTable\_m”/>

</field>

</fields>

</parameter>

Properties :

* The class specified in the basic-type must be IpAddressType because the SNMP type is IpAddress.
* No length element may be specified in the basic-type because we have 1 ip-address, not a list of 4 integers.
* The default value is written as &quot;0.0.0.0&quot; because we want it to be “0.0.0.0” in C.
  + 1. Enumeration pattern
       1. When to use ?

Use this pattern for parameters where the operator must select from a limited set of possibilities and where each of these possibilities can be given a name. Never use this pattern for boolean parameters : use the boolean pattern instead (see section 5.1.3).

* + - 1. Basic construction

In a parameter type XML file:

<parameter-type name=”Equip::ShubAdminStatus”>

<help>the admin status for shub related equipment</help>

<alternative-field name=”” basic-type=”SignedInteger”>

<alternatives>

<alternative identifier=”up” help=”the equipment is up”

value=”(long)E\_slotAdminStatus\_up”/>

<alternative identifier=”down” help=”the equipment is down”

value=”(long) E\_slotAdminStatus\_down”/>

<alternative identifier=”auto-up”

help=”the equipment is autonomously brought up”

value=”(long)E\_slotAdminStatus\_autoup”/>

<alternative identifier=”testing”

help=”the system is testing the equipment”

value=”(long) E\_slotAdminStatus\_testing” access=”read”/>

</alternatives>

</alternative-field>

</parameter-type>

In a node XML file :

<parameter name=”admin-status” parameter-type=”Equip::ShubAdminStatus”>

<help>the admin status of the equipment</help>

<optional default=”&quot;auto-up&quot;”/>

<fields>

<field name=””>

<mib-var tree-node=”NODE\_systemAdminStatus”

table-name=”SnmpAgent::localScalarTable\_m”/>

</field>

</fields>

</parameter>

Properties :

* Parameter is named
* A name is assigned to each alternative. The name is given in lower-case and words are separated by underscores. Names should be maximum 17 characters long.
* The help linked to the alternatives is completely in lower case. In practise, the maximum number of characters is no longer limited. However, we propose a soft-limit of 3x56 characters.
* A basic type is assigned to the alternative-field. The basic type typically does not have any restrictions as the possible values an operator can enter are already restricted by the pattern itself.
* The values given to each of the alternatives is given in the format as the user must type it in for the specified basic type. Knowing that E\_slotAdminStatus\_up is equal to 1, we could have written the value as &quot;1&quot;
* The value testing can not be entered by the operator because the access is read. This value can be shown in the output of a **show** or **info** command.
* A default value can be specified or not, but if the default value is specified, it must be the name of one of the alternatives.

Verification via CLI :

* Enter : **?**

**[no] admin-status : the admin status of the equipment**

**default = auto-up**

* Enter : **admin-status ?**

**up ! the equipment is up**

**down ! the equipment is down**

**auto-up ! the equipment is autonomously brought up**

Attention points :

* The help of the parameter and the default value are shown for **?**
* The name and help of each of the alternatives that the operator can enter are shown for **admin-status ?**
* The name and help of the alternative testing is not shown.
  + - 1. Variant : alternatives and builtin-type

The following construct allows the operator to enter an alternative (local) or a builtin-type (an ip-address).

In a parameter type XML file:

<parameter-type name=”Swmgt::HostAddress”>

<help>the address of a host</help>

<alternative-field name=”” basic-type=”Ip::V4Address”>

<alternatives>

<alternative identifier=”local” help=”the local host”

value=”&quot;127.0.0.1&quot;”/>

<alternative identifier=”shub” help=”the service hub”

value=”&quot;127.0.0.3&quot;”/>

<default help=”the ip-v4 address of a host”/>

</alternatives>

</alternative-field>

</parameter-type>

In a node XML file :

<parameter name=”from” parameter-type=”Swmgt::HostAddress”>

<help>the address of the host where the file must be copied from</help>

<optional default=”&quot;local&quot;”/>

<fields>

<field name=””>

<mib-var tree-node=”NODE\_fileCopyFrom”

table-name=SnmpAgent::localScalarTable\_m”/>

</field>

</fields>

</parameter>

Properties :

* The default construct in the alternative-field indicates that if not one of the alternatives is matched, the basic type must be used.
* In this construct it can be necessary to use a basic type with restrictions.
* A basic type is assigned to the alternative-field. The basic type typically does not have any restrictions as the possible values an operator can enter are already restricted by the pattern itself.
* The value of the alternatives must be accepted by the basic type : the operator should be able to enter both local and 127.0.0.1.

Verification via CLI :

* Enter : **?**

**[no] from : the address of the host where the file must be copied from**

**default = local**

* Enter : **from ?**

**local ! the local host**

**shub ! the service hub**

**<Ip::V4Address> ! a IPv4 address**

Attention points :

* The help of the parameter and the default value are shown for **?**
* The name and help of each of the alternatives that the operator can enter are shown for **admin-status ?**
* The name and help of the basic type is shown for the alternative default.
  + - 1. Variant : alternatives and read-only builtin-type

This is identical to the previous case, but the default has access equal to read :

<parameter-type name=”Swmgt::HostAddress”>

<help>the address of a host</help>

<alternative-field name=”” basic-type=”Ip::V4Address”>

<alternatives>

<alternative identifier=”local” help=”the local host”

value=”&quot;127.0.0.1&quot;”/>

<alternative identifier=”shub” help=”the service hub”

value=”&quot;127.0.0.3&quot;”/>

<default help=”the ip-v4 address of a host” access=”read”/>

</alternatives>

</alternative-field>

</parameter-type>

The result is that the operator only can enter local and shub but no other ip-address. Other ip-address can be displayed via a **show** or an **info** command.

* + 1. Boolean pattern
       1. When to use ?

Use this pattern **always** for boolean parameters (example : true/false, enable/disable, activate/deactivate). **Never** use this pattern when a parameter can have two values and the two values are not logical opposites (example : “snmp-v1” and “snmp-v2”).

* + - 1. Basic Construction

In a parameter-type XML file :

<parameter-type name=”Trap::ColdStartTrap”

<help>controls the sending of the cold-start trap</help>

<alternative-field name=”” basic-type=”SignedInteger”>

<alternatives>

<alternative identifier=”cold-start-trap”

help=”enable sending of a cold-start trap”

value=”(long)E\_AsamColdStartTrap\_Enable”/>

<alternative identifier=”no-cold-start”

help=”disable sending of a cold-start trap”

value=”(long)E\_AsamColdStartTrap\_Disable”/>

</alternatives>

</alternative-field>

</parameter-type>

In a node XML file :

<parameter uname=”cold-start-trap” parameter-type=”Trap::ColdStartTrap”>

<help> enable sending of a cold-start trap </help>

<optional default=”no-cold-start”/>

<fields>

<field name=””>

<mib-var tree-node=”NODE\_asamColdStartTrap”

table-name=”SnmpAgent::localScalarTable\_m”/>

</field>

</fields>

</parameter>

Properties :

* Parameter is unnamed
* The name of the positive alternative does not contain the words “true”, “enable”, ….)
* Name of parameter is equal to the positive alternative (true/activate/enable)
* The default value is mandatory and is equal to the negative alternative (false/deactivate/disable)
* Help of the parameter is equal to the help of the positive alternative

Verification via CLI :

* Enter : **?**

**[no] cold-start-trap : enable sending of a cold-start trap**

* Enter : **no ?**

**cold-start-trap : enable sending of a cold-start-trap**

Attention points :

* same name and help are shown in the **?** and the **no ?** case
* help and name of the negative alternative are never visible to the operator
  + 1. Combined parameters
       1. When to use ?

Use this pattern **always** when the operator needs to enter a number of parameters that always need to be entered together.

* + - 1. Basic Construction

In a parameter-type XML file :

<parameter-type name=”Equip::Slot”

<help>the physical position of the slot</help>

<fields>

<field name="rack" basic-type="Eqpt::Rack" separator="/"/>

<field name="shelf" basic-type="Eqpt::Shelf" separator="/"/>

<field name="slot" basic-type="Equipm::Slot"/>

</fields>

</parameter-type>

The node XML file contains :

<parameter name=”nt-slot” parameter-type=”Equip::Slot”>

<help>the slot in which the nt is plugged in</help>

<optional default=”&quot;1/1/2&quot;”/>

<fields>

<field name=”rack”>

<mib-var tree-node=”NODE\_ntPostionRack”

table-name=”SnmpAgent::localScalarTable\_m”/>

</field>

<field name=”shelf”>

<mib-var tree-node=”NODE\_ntPostionShelf”

table-name=”SnmpAgent::localScalarTable\_m”/>

</field>

<field name=”slot”>

<mib-var tree-node=”NODE\_ntPostionSlot”

table-name=”SnmpAgent::localScalarTable\_m”/>

</field>

</fields>

</parameter>

Properties :

* The rack must be entered first and is separated from the shelf with a slash.
* The slot must be entered after the shelf and is separated from the shelf with a slash.
* The default value of the parameter is the string 1/1/2 which is a correct combination of all fields.
* Each field can get in the node XML file different properties.
  + - 1. Variant : limited number of combinations

In the following example must the operator be able to enter following combinations (the combination **any-any** is not allowed) :

* <*mac-address*>**-**<*mac-address*> (version 1)
* **any-**<*mac-address*> (version 2)
* <*mac-address*>**-any** (version 3)

The parameter-types XML file contains :

<parameter-type name="Sys::MacFilter">

<help>specifies the mac-addresses to be filtered</help>

<fields>

<alternative-field name="source" basic-type="Sys::MacAddr" separator="-">

<alternatives>

<alternative identifier="any" help="all source MAC address"

value="&quot;00:00:00:00:00:00&quot;" version="2"/>

<default help="source mac address" version="1,3"/>

</alternatives>

</alternative-field>

<alternative-field name="destination" basic-type="Sys::MacAddr">

<alternatives>

<alternative identifier="any" help="all destination MAC address"

value="&quot;00:00:00:00:00:00&quot;" version="3"/>

<default help="destination mac address" version="1,2"/>

</alternatives>

</alternative-field>

</fields>

</parameter-type>

Properties :

* Each alternative gets a version tag to indicate in which version of the parameter-type it participates. A parameter-type version is thus a given combination fields.
* The **any** alternative of the alternative-field **source** only participates in version 2.
* The default alternative (the <*mac-address*>) of the alternative-field **source** participates in versions 1 and 3. The version numbers are in the list separated by commas
* The **any** alternative of the alternative-field **destination** only participates in version 3.
* The default alternative (the <*mac-address*>) of the alternative-field **destination** participates in versions 1 and 2. The version numbers are in the list separated by commas.
  + - 1. Variant : variable number of fields

The operator must be able to enter following values :

* **lt:**<*Eqpt::Rack*>/<*Eqpt::Shelf*>/<*Eqpt::Slot*> (version 1)
* **network:**<*Shub::NetworkPort*> (version 2)
* **nt** (version 3)

The parameter-type XML file contains :

<parameter-type name="Shub::Port">

<help>the port identification</help>

<fields>

<alternative-field name="" basic-type="Shub::DataPort" separator=":">

<alternatives>

<alternative identifier="lt" help="port of the line board" value="1L" version="1"/>

<alternative identifier="network" help="network port" value="18L" version="2"/>

<alternative identifier="nt" help="port of the nt" value="17L" version="3"/>

</alternatives>

</alternative-field>

<field name="rack" basic-type="Eqpt::Rack" separator="/" version="1"/>

<field name="shelf" basic-type="Eqpt::Shelf" separator="/" version="1"/>

<field name="slot" basic-type="Eqpt::Slot" separator="/" version="1"/>

<field name="network" basic-type="Shub::NetworkPort" version="2"/>

</fields>

</parameter-type>

Properties :

* The system does not prompt the operator for the separator : after **nt** because there is no next field.
* The system does not prompt the operator for the separator / after <*Eqpt::Slot*> because there is no next field.
* Fields **rack**, **shelf** and **slot** are skipped for versions 2 and 3.
* Field **networ**k is skipped for versions 1 and 3.
  + - 1. Variant : alternatives and read-only builtin-type

In the following example must the operator be able to enter :

* <*Vlan::UVlanIndex*>
* **stacked:**<*Vlan::SVlanIndex*>:<*Vlan::CVlanIndex*>

Special in this example is that the 0L which corresponds to the alternative **stacked** may not be entered in case of <*Vlan::UVlanIndex*>

The basic-type XML file contains :

<basic-type name="Vlan::UVlanIndex" class="UnsignedIntegerType">

<help>unstacked vlan identity </help>

<helprange>[1...4093]</helprange>

<restrictions>

<range min-included="1" max-included="4093" action="accept" type="unsigned"/>

<range min-included="0" max-included="0" action="accept" type="unsigned"

can-be-entered="false"/>

</restrictions>

</basic-type>

The parameter-type XML file contains :

<parameter-type name="Vlan::StackedVlan">

<help>generic vlan identity</help>

<fields>

<alternative-field name="unstacked" basic-type="Vlan::UVlanIndex" separator=":">

<alternatives>

<alternative identifier="stacked" help="stacked vlan identity" value="0L" version="2"/>

<default help="unstack vlan identity" version="1"/>

</alternatives>

</alternative-field>

<field name="svlan" basic-type="Vlan::SVlanIndex" separator=":" version="2"/>

<field name="cvlan" basic-type="Vlan::CVlanIndex" version="2"/>

<field name="" basic-type="Vlan::StackedVlan" version="32"/>

</fields>

</parameter-type>

Properties :

* A range from 0L to 0L but with attribute can-be-entered equal to false is added to the basic-type.
  + - 1. Variant : multiple variants with same version

The operator must be able to enter :

* **none**
* **md5:**<*Password*>
* **sha:**<*Password*>

The parameter XML file contains then :

<parameter-type name="Aaa::Authentication">

<help>the required authentication</help>

<fields>

<alternative-field name="" basic-type="SignedInteger" separator=":">

<alternatives>

<alternative identifier="none" help="no authentication is required" value="1L" version="1"/>

<alternative identifier="md5" help="use MD5 based hashing" value="2L" version="2"/>

<alternative identifier="sha" help="use SHA based authentication" value="3L" version="2"/>

</alternatives>

</alternative-field>

<field name="password" basic-type=”Password" version="2"/>

</fields>

</parameter-type>

Properties :

* The alternatives **md5** and **sha** use the same version number.
  1. Node patterns
     1. Scalar config node
        1. When to use ?

Use this pattern always for a **configure** sub-node that is used to manage scalar variables. Note that a **configure** sub-node can not manage a mix of scalar variables and table variables.

* + - 1. Basic construction

The node XML file looks like :

<?xml version="1.0" encoding="UTF-8" ?>

<nodes xmlns:xsi=<http://www.w3.org/2001/XMLSchema-instance>

xsi:noNamespaceSchemaLocation=[file:///cm4/tools/BELL/xml/xmlCommon/node.xsd](file:///F:\cm4\tools\BELL\xml\xmlCommon\node.xsd)

language="CLI">

<node name="sntp" access="AccessRights::swmgnt\_rw">

<help>configure sntp</help>

<man-page file="CLI\_System/manConfigSntp.xml" />

<default-command>

<command handler-type="configure" class="CommandDefinition"/>

</default-command>

<commands>

<command handler-type="info" class="CommandDefinition" />

<command handler-type="help" class="CommandDefinition" />

<command handler-type="tree" class="CommandDefinition" />

</commands>

<parameters>

<parameter name=”server” parameter-type=”Ip::V4Address”>

<help>the ip-address of the sntp server</help>

<optional default=”&quot;0.0.0.0&quot;”/>

<fields>

<field name=””>

<mib-var tree-node=”NODE\_ AsamSNTPServerIpAddress”

table-name=”SnmpAgent::localScalarTable\_m”/>

</field>

</fields>

</parameter>

<parameter name=”polling-rate” parameter-type=”Sys::SntpPollingRate”>

<help>the rate to poll the sntp-server</help>

<optional default=”300L”/>

<fields>

<field name=””>

<mib-var tree-node=”NODE\_AsamSNTPPollingRate”

table-name=”SnmpAgent::localScalarTable\_m”/>

</field>

</fields>

</parameters>

</node>

</nodes>

The man page XML looks like :

<?xml version="1.0" encoding="UTF-8" ?>

<manpage xmlns:xsi=<http://www.w3.org/2001/XMLSchema-instance>

xsi:noNamespaceSchemaLocation="file:///cm4/tools/BELL/xml/xmlCommon/manpage.xsd" language="CLI">

<userguide>

<title>Alcatel iSAM CLI User Guide</title>

<author>Frank Denissen</author>

<chapter title="Sntp Configuration Command">

<chapter ref="cmd\_descr">

<p>This command allows the operator to configure the parameters of SNTP, the Simple Network Time Protocol.</p>

</chapter>

<chapter ref="usr\_level" />

<chapter isCommand="true" />

<chapter ref="cmd\_output" />

</chapter>

</userguide>

</manpage>

Properties :

* The light blue part must be present in each node XML file
* The dark blue part must be present in each man page XML file
* The name of the node is maximum 22 characters long, completely in lower-case and the words are separated by dashes.
* The domain to which an operator must belong to execute this command is here software management. The value is one of the values you can find in **CLI\_Agent/AccessRights.hpp**. Take always the value with **\_rw** at the end.
* The help of a sub-node of the **configure** top-node always starts with the string “**configure**”. It is always in written in lower-case.
* Each node has its own man page XML. It is typically located in the same directory and the name of the man-page starts with **man**.
* The name of the default command for a sub-node of the **configure** top-node that is not an intermediate node, is always **configure**.
* The list of environment commands for a sub-node of the configure top-node is always : tree, info and help.
* The man page contains the name of the author that made the last changes to the man-page.
* Each word in the title starts with a capital.
* The title of sub-nodes of the configure top-node ends with **Configuration Command**.
* The chapter cmd\_descr contains at least one paragraph that gives a brief explanation on the command. It does NOT explain the individual parameters and does NOT contain screen dumps of commands.

Detailed information on how to write man pages can be found in [5]

* + 1. Static config node
       1. When to use ?

Use this pattern always for a **configure** sub-node that is used to manage instances of a static table. A static table is a table in which the operator can not create new entries : all entries are pre-defined by the system or created autonomously by the system.

* + - 1. Basic construction

The MIB looks as follows :

eqptHolderTable OBJECT-TYPE

SYNTAX SEQUENCE OF EqptHolderEntry

ACCESS not-accessible

STATUS mandatory

DESCRIPTION

"Equipment holder table, representing the system, the racks and

the shelves.

This table contains one row for the system, one row per rack,

and one row per shelf.

ALCATEL NOTE:

SUPPORT: YES TABLESIZE: DEP as described above"

::= { asamEquipmentMIB 2 }

eqptHolderEntry OBJECT-TYPE

SYNTAX EqptHolderEntry

ACCESS not-accessible

STATUS mandatory

DESCRIPTION

"An entry of the equipment holder table.

ALCATEL NOTE:

SUPPORT: YES"

INDEX { eqptHolderId }

::= { eqptHolderTable 1 }

We create a CLI\_Equipment/export/equipment\_tables.hpp with following content :

#ifndef \_\_EQUIPMENT\_TABLES\_HPP\_

#define \_\_EQUIPMENT\_TABLES\_HPP\_

#ifndef \_\_MIBTABLEDEFINITION\_HPP\_

#include “CLI\_Agent/MibTableDefinition.hpp”

extern MibTableDefinition eqptBoardTable;

#endif

We create a CLI\_Equipment/equipment\_tables.cpp file with following content :

#ifndef \_\_EQUIPMENT\_TABLES\_HPP\_

#include “CLI\_Equipment/equipment\_tables.hpp”

#endif

#ifndef \_\_SNMPAGENT\_HPP\_

#include “CLI\_Agent/SnmpAgent.hpp”

#endif

namespace Cli {

const MibTableDefinition::MibKeyDefinition eqptBoardTableKeys = {

{“holder-index”, false}

};

MibTableDefinition eqptBoardTable(“eqptBoardTable”,

&SnmpAgent::localSnmpAgent\_m,

eqptBoardTableKeys,

sizeof(eqptBoardTableKeys)/sizeof(eqptBoardTableKeys[0]));

} // end namespace Cli

The node XML file looks like :

<?xml version="1.0" encoding="UTF-8" ?>

<nodes xmlns:xsi=<http://www.w3.org/2001/XMLSchema-instance>

xsi:noNamespaceSchemaLocation=[file:///cm4/tools/BELL/xml/xmlCommon/node.xsd](file:///F:\cm4\tools\BELL\xml\xmlCommon\node.xsd)

language="CLI">

<include file=”CLI\_Equipment/equipment\_tables.hpp”/>

<node name="holder" access="AccessRights::equip\_rw">

<help>configure a holder</help>

<man-page file="CLI\_Equipment/manConfigHolder.xml" />

<default-command>

<command handler-type="configure" class="CommandDefinition"/>

</default-command>

<commands>

<command handler-type="info" class="CommandDefinition" />

<command handler-type="help" class="CommandDefinition" />

<command handler-type="tree" class="CommandDefinition" />

</commands>

<parameters>

<res-id uname=”holder” parameter-type=”Equip::Holder”>

<help>the slot number</help>

<fields>

<field name=””>

<mib-index name=”holder-index”/>

</field>

</fields>

</res-id>

<parameter name=”planned-type” parameter-type=”Equip::PlannedType”>

<help>the type of the board planned to be inserted in this slot</help>

<optional default=”&quot;not-planned&quot;”/>

<fields>

<field name=””>

<mib-var tree-node=”NODEeqptSlotPlannedType” table-name=”eqptBoardTable”/>

</field>

</fields>

</parameter>

</parameters>

</node>

</nodes>

Properties :

* The location in the table definition eqptBoardTable is SnmpAgent::localSnmpAgent\_m because the table is handled by the ASAM part of ISAM.
* The MIB indicates that the table has one index. In the table definition eqptBoardTable we specify that the field that contains the index will be tagged with the name holder-index. Holder-index corresponds to eqptHolderId in the MIB. It is recommended not to use the MIB name as tag for the index to make it easier to combine different tables in 1 command (see later).
* The false in the key definition indicates that the corresponding index is in the SNMP coding not preceeded by a length indicator. All fixed length types (like **GAUGE**, **COUNTER**, **INTEGER**, …) or all fixed-length types derived from variable length types (like **OBJECT ID** and **OCTET STRING**) are not preceeded by a length indicator. Example : use false for **OCTET STRING (SIZE (4..4))**.
* The header file equipment\_tables.hpp is included in the node XML file
* A res-id element is used to indicate that this parameter identifies the instance. The combination of all res-id must identify 1 instance in a unique way : two different sets of res-id may never identify the same instance.
* The first res-id of a node is always unnamed.
* The field with name **“”** of res-id **holder** is tagged to hold the value for the index holder-index.
* The name of the table eqptBoardTable is specified as value for the attribute **table-name** of the element **mib-var**.
* One <*Name*>**\_tables.hpp** and <*Name*>**\_tables.cpp** can contain the definitions of many tables.
  + - 1. Variant : the index is of a variable length type

The MIB looks like :

communityTable OBJECT-TYPE

SYNTAX SEQUENCE OF communityEntry

ACCESS not-accessible

STATUS mandatory

DESCRIPTION

"Community holder table, representing the list of community

strings used as identification for SNMP"

::= { snmpMIB 2 }

communityEntry OBJECT-TYPE

SYNTAX CommunityEntry

ACCESS not-accessible

STATUS mandatory

DESCRIPTION

"An entry of the community table.

ALCATEL NOTE:

SUPPORT: YES"

INDEX { communityName }

::= { communityTable 1 }

communityName OBJECT-TYPE

SYNTAX DisplayString (SIZE (1..64))

ACCESS not-accessible

STATUS mandatory

DESCRIPTION

"The name of the community"

::= { communityEntry 1 }

We add following lines CLI\_Transport/transport\_tables.cpp file with following content :

const MibTableDefinition::MibKeyDefinition communityTableKeys = {

{“name”, true}

};

MibTableDefinition communityTable(“communityTable”,

&SnmpAgent::localSnmpAgent\_m,

communityTableKeys,

sizeof(communityTableKeys)/sizeof(communityTableKeys[0]));

The node XML file contains :

<res-id uname=”name” parameter-type=”Snmp::Community”>

<help>the community string</help>

<fields>

<field name=””>

<mib-index name=”name”/>

</field>

</fields>

</res-id>

Properties :

* The key definition contains now true
  + - 1. Variant : the table is handled by LANX

The **CLI\_VLAN/vlan\_tables.cpp** contains :

const MibTableDefinition::MibKeyDefinition dot1dPortPriorityTableKeys[] = {

{"dot1dBasePort", false}

};

MibTableDefinition lanx\_dot1dPortPriorityTable("lanx\_dot1dPortPriorityTable",

&SnmpAgent::lanxSnmpAgent\_m,

dot1dPortPriorityTableKeys,

sizeof(dot1dPortPriorityTableKeys)/sizeof(dot1dPortPriorityTableKeys[0]));

MibTableDefinition dot1dPortPriorityTable("dot1dPortPriorityTable",

&SnmpAgent::localSnmpAgent\_m,

dot1dPortPriorityTableKeys,

sizeof(dot1dPortPriorityTableKeys)/sizeof(dot1dPortPriorityTableKeys[0]));

The node XML file contains :

<parameter name="default-priority" parameter-type="Shub::PortPriority">

<help>default ingress user priority</help>

<optional default="&quot;0&quot;"/>

<fields>

<field name="">

<mib-var tree-node="NODEdot1dPortDefaultUserPriority"

table-name="dot1dPortPriorityTable" lanx="true"/>

</field>

</fields>

</parameter>

Properties :

* The table name in the vlan\_tables.cpp is always preceeded by lanx\_.
* The location in the table definition is SnmpAgent::lanxSnmpAgent\_m
* The table name in the node XML file is written without the lanx\_ prefix
* The attribute lanx of the mib-var element has the value true
* The key definitions are shared between the two table definitions (the first for LANX, the second for ASAM).
  + - 1. Variant : the table has multiple indices

The MIB has following contents :

domainIpPoolTable OBJECT-TYPE

SYNTAX SEQUENCE OF DomainIpPoolEntry

ACCESS not-accessible

STATUS mandatory

DESCRIPTION

" This is the (conceptual) table for listing the mapping between the Ip Pool

Name and Ip Pool Index."

::= { ipPoolMIB 3 }

domainIpPoolEntry OBJECT-TYPE

SYNTAX DomainIpPoolEntry

ACCESS not-accessible

STATUS mandatory

DESCRIPTION

" This is the (conceptual) entry used for Mapping the Ip Pool Name

to the Pool Index."

INDEX { domainIpPoolVrf, domainIpPoolName}

::= { domainIpPoolTable 1 }

The CLI\_AAA/aaa\_tables.cpp file contains :

const MibTableDefinition::MibKeyDefinition domainIpPoolTableKeys[] = {

{"vrf", false},

{"name", true}

};

MibTableDefinition domainIpPoolTable("domainIpPoolTable",

&SnmpAgent::localSnmpAgent\_m,

domainIpPoolTableKeys,

sizeof(domainIpPoolTableKeys)/sizeof(domainIpPoolTableKeys[0]));

The node XML file contains :

<?xml version="1.0" encoding="UTF-8" ?>

<nodes xmlns:xsi=<http://www.w3.org/2001/XMLSchema-instance>

xsi:noNamespaceSchemaLocation=[file:///cm4/tools/BELL/xml/xmlCommon/node.xsd](file:///F:\cm4\tools\BELL\xml\xmlCommon\node.xsd)

language="CLI">

<include file=”CLI\_AAA/aaa\_tables.hpp”/>

<node name="ip-pool" access="AccessRights::aaa\_rw">

<help>configure a ip-address pool</help>

<man-page file="CLI\_AAA/manConfigIpAddressPool.xml" />

<default-command>

<command handler-type="configure" class="CommandDefinition"/>

</default-command>

<commands>

<command handler-type="info" class="CommandDefinition" />

<command handler-type="help" class="CommandDefinition" />

<command handler-type="tree" class="CommandDefinition" />

</commands>

<parameters>

<res-id uname=”name” parameter-type=”Aaa::IpPoolName”>

<help>the name of the ip-pool</help>

<fields>

<field name=””>

<mib-index name=”name”/>

</field>

</fields>

</res-id>

<res-id name=”vrf” parameter-type=”Aaa::Vrf”>

<help>the vrf number</help>

<fields>

<field name=””>

<mib-index name=”vrf”/>

</field>

</fields>

</res-id>

<parameter name="priority" parameter-type="SignedInteger">

<help>priority associated with this IP pool</help>

<fields>

<field name="">

<mib-var tree-node="NODEdomainIpPoolPriority" table-name="domainIpPoolTable"/>

</field>

</fields>

</parameter>

</parameters>

<fields>

<field name=”in” basic-type=”Aaa::PoolPriority”

access=”CommandFieldDefinition::mayBeReadDuringIn\_c”>

<mib-var tree-node="NODEdomainIpPoolPriority" table-name="domainIpPoolTable"/>

</field>

</fields>

</node>

</nodes>

Properties :

* The order in which the indices are given in the table definition must be the order found in the MIB.
* It is not mandatory that the order in which the res-id are given is the same as in the MIB. Here the choice was made to put the name first because the corresponding command looks more natural for the operator : **ip-pool** <*name*> **vrf** <*vrf*>
* The first res-id is unnamed.
* The second res-id and all following ones are named.
* For tables for which the system does not guarantee that an instance exists for all possible combinations of the indices, must a special node field be added to the node. This field is needed here because the system does not guarantee that an instance exists for each arbitrary **vrf** or **name** combination.
* This auxiliary field will be read to verify if the node instance exists or not. That is why the access is mayBeReadDuringIn.
* The MIB variable related to the special node field is chosen arbitrary : it can be any of the columns of the MIB table.
* Note that a **basic-type** defines the auxiliary field and not a **parameter-type**. This is because an auxiliary field can never be entered by the operator.
  + - 1. Variant : an index has a fixed value

Suppose that the index **vrf** of the previous example has a fixed value **1**. The node XML changes then as follows :

<?xml version="1.0" encoding="UTF-8" ?>

<nodes xmlns:xsi=<http://www.w3.org/2001/XMLSchema-instance>

xsi:noNamespaceSchemaLocation=[file:///cm4/tools/BELL/xml/xmlCommon/node.xsd](file:///F:\cm4\tools\BELL\xml\xmlCommon\node.xsd)

language="CLI">

<include file=”CLI\_AAA/aaa\_tables.hpp”/>

<node name="ip-pool" access="AccessRights::aaa\_rw">

<help>configure a ip-address pool</help>

<man-page file="CLI\_AAA/manConfigIpAddressPool.xml" />

<default-command>

<command handler-type="configure" class="CommandDefinition"/>

</default-command>

<commands>

<command handler-type="info" class="CommandDefinition" />

<command handler-type="help" class="CommandDefinition" />

<command handler-type="tree" class="CommandDefinition" />

</commands>

<parameters>

<res-id uname=”name” parameter-type=”Aaa::IpPoolName”>

<help>the name of the ip-pool</help>

<fields>

<field name=””>

<mib-index name=”name”/>

</field>

</fields>

</res-id>

<parameter name="priority" parameter-type="Aaa::PoolPriority">

<help>priority associated with this IP pool</help>

<fields>

<field name="">

<mib-var tree-node="NODEdomainIpPoolPriority" table-name="domainIpPoolTable"/>

</field>

</fields>

</parameter>

</parameters>

<fields>

<field name=”vrf” basic-type=”Aaa::Vrf”

access=”CommandFieldDefinition::mayBeUsedDuringCreate\_c|

CommandFieldDefinition::mayBeUsedDuringPrepare\_c|

CommandFieldDefinition::mayBeUsedDuringModify\_c|

CommandFieldDefinition::mayBeUsedDuringCommit\_c|

CommandFieldDefinition::mayBeUsedDuringIn\_c|

CommandFieldDefinition::mayBeUsedDuringDisplay\_c” default=”1L”>

<mib-index name=”vrf”/>

</field>

<field name=”in” basic-type=”Aaa::PoolPriority”

access=”CommandFieldDefinition::mayBeReadDuringIn\_c”>

<mib-var tree-node="NODEdomainIpPoolPriority" table-name="domainIpPoolTable"/>

</field>

</fields>

</node>

</nodes>

Properties :

* The **res-id** for the **vrf** is removed
* It is replaced by a new auxiliary field vrf with a fixed value 1L.
* The new field may be present in each phase because an index is always needed to access a particular instance. That is why the access field is a combination of all mayBeUsedDuring<*Phase*> values.
* The mib-index construct now indicates that the auxiliary field is to be used as index.
  + - 1. Variant : two tables in one command

In the following example it is logical to configure in one command the port-type and the admin-status. Both parameters are however controlled by the different MIB tables :

issPortCtrlTable OBJECT-TYPE

SYNTAX SEQUENCE OF IssPortCtrlEntry

ACCESS not-accessible

STATUS mandatory

DESCRIPTION

"A table to control the port specific parameters of the device like speed,

duplex mode, etc."

::= { issConfigControl 2 }

issPortCtrlEntry OBJECT-TYPE

SYNTAX IssPortCtrlEntry

ACCESS not-accessible

STATUS mandatory

DESCRIPTION

"An entry appears in this table for each interface in the system.

Index to the table is the interface index of the port."

INDEX {issPortCtrlIndex}

::= { issPortCtrlTable 1 }

issPortCtrlIndex OBJECT-TYPE

SYNTAX INTEGER (0..65535)

ACCESS not-accessible

STATUS mandatory

DESCRIPTION

"Interface index of the port for which the configuration

in this entry applies."

::= { issPortCtrlEntry 1 }

InterfaceIndex ::= SYNTAX INTEGER (1..2147483647)

ifMainTable OBJECT-TYPE

SYNTAX SEQUENCE OF IfMainEntry

ACCESS not-accessible

STATUS mandatory

DESCRIPTION

"A list of all the interface entries in the system.

This table contains objects which are applicable to all

types of interfaces in the system. This table is a

proprietary extension to the standard ifTable and

ifXTable. The index to this table has the semantics of

the MIB-2 ifIndex."

::= { if 4 }

ifMainEntry OBJECT-TYPE

SYNTAX IfMainEntry

ACCESS not-accessible

STATUS mandatory

DESCRIPTION

"An entry containing management information applicable

to a particular interface."

INDEX {ifMainIndex}

::= { ifMainTable 1 }

ifMainIndex OBJECT-TYPE

SYNTAX InterfaceIndex

ACCESS not-accessible

STATUS mandatory

DESCRIPTION

"A unique value, greater than zero, for each

interface. This object is identical to the ifIndex

of the standard MIB-2 ifTable."

::= { ifMainEntry 1 }

Notes :

* A mandatory condition to be able to use two tables in the same command is that both tables require the same values for the indices.
* It is not important that the range of ifMainIndex (1..2147483647) is larger than the range of issPortCtrlIndex (0..65535) because the possible values will be limited by the parameter type Shub::NetworkPort.
* Do not use this construct when both entries must be created in the command. This is not supported by CLI. See section 1.2.4.

So we add following lines to CLI\_System/system\_tables.cpp :

const MibTableDefinition::MibKeyDefinition issPortCtrlKeys[] = {

{"port", false}

};

MibTableDefinition lanx\_issPortCtrlTable("lanx\_issPortCtrlTable",

&SnmpAgent::lanxSnmpAgent\_m,

issPortCtrlKeys,

sizeof(issPortCtrlKeys)/sizeof(issPortCtrlKeys[0]));

const MibTableDefinition::MibKeyDefinition ifMainTableKeys[] = {

{"port", false}

};

MibTableDefinition lanx\_ifMainTable("lanx\_ifMainTable",

&SnmpAgent::lanxSnmpAgent\_m,

ifMainTableKeys,

sizeof(ifMainTableKeys)/sizeof(ifMainTableKeys[0]));

The node XML file becomes then :

<?xml version="1.0" encoding="UTF-8"?>

<nodes xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="file:///cm4/tools/BELL/xml/xmlCommon/node.xsd"

language="CLI">

<include file="./CLI\_System/system\_tables.hpp"/>

<node name="port" access="AccessRights::all\_rw">

<help>configure a network port</help>

<man-page file="./CLI\_System/manIssPortCtrl.xml"/>

<default-command>

<command handler-type="configure" class="CommandDefinition"/>

</default-command>

<commands>

<command handler-type="info" class="CommandDefinition"/>

<command handler-type="help" class="CommandDefinition"/>

<command handler-type="tree" class="CommandDefinition"/>

</commands>

<parameters>

<res-id uname="network-port" parameter-type="Shub::NetworkPort">

<help>identifier of the port interface</help>

<fields>

<field name="">

<mib-index name="port"/>

</field>

</fields>

</res-id>

<parameter name="port-type" parameter-type="Sys::Type">

<help>the type of port interface</help>

<optional default="&quot;unused&quot;"/>

<fields>

<field name="">

<mib-var tree-node="NODEissPortCtrlType" table-name="issPortCtrlTable" lanx="true"/>

</field>

</fields>

</parameter

<parameter name="admin-status" parameter-type="Shub::AdminStatus">

<help>administrative state of the port</help>

<optional default="&quot;down&quot;"/>

<fields>

<field name="">

<mib-var tree-node="NODEifMainAdminStatus" table-name="ifMainTable" lanx=”true”/>

</field>

</fields>

</parameter>

</parameters>

</node>

</nodes>

Properties :

* In the **\_tables.cpp** file have both tables the same name (port) for the index while the names of the indices are different in the MIB (issPortCtrlIndex and ifMainIndex).
* In the node XML only one field exists with mib-index port.
* The fields of both parameters refer to a different table.
  + 1. Dynamic config node
       1. When to use ?

Use this pattern always when a **configure** sub-node is used for a table in which the operator can create new instances. This is typically the case when the MIB table contains a column of type RowStatus.

* + - 1. Basic construction

The MIB file contains :

dot1qVlanStaticTable OBJECT-TYPE

SYNTAX SEQUENCE OF Dot1qVlanStaticEntry

ACCESS not-accessible

STATUS mandatory

DESCRIPTION

"A table containing static configuration information for

each VLAN configured into the device by (local or

network) management. All entries are permanent and will

be restored after the device is reset."

::= { dot1qVlan 3 }

dot1qVlanStaticEntry OBJECT-TYPE

SYNTAX Dot1qVlanStaticEntry

ACCESS not-accessible

STATUS mandatory

DESCRIPTION

"Static information for a VLAN configured into the

device by (local or network) management."

INDEX { dot1qVlanIndex }

::= { dot1qVlanStaticTable 1 }

Dot1qVlanStaticEntry ::=

SEQUENCE {

dot1qVlanStaticName

SnmpAdminString,

dot1qVlanStaticEgressPorts

PortList,

dot1qVlanForbiddenEgressPorts

PortList,

dot1qVlanStaticUntaggedPorts

PortList,

dot1qVlanStaticRowStatus

RowStatus

}

The node XML file looks like :

<?xml version="1.0" encoding="UTF-8"?>

<nodes xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="file:///cm4/tools/BELL/xml/xmlCommon/node.xsd"

language="CLI">

<node name="id" access="AccessRights::vlan\_rw" is-dynamic="true">

<help>configure a vlan</help>

<man-page file="CLI\_VLAN/manConfigureVlanid.xml"/>

<default-command>

<command handler-type="configure" class="CommandDefinition"/>

</default-command>

<commands>

<command handler-type="info" class="CommandDefinition"/>

<command handler-type="help" class="CommandDefinition"/>

<command handler-type="tree" class="CommandDefinition"/>

</commands>

<parameters>

<res-id uname="vlan-id" parameter-type="Vlan::VlanId">

<help>vlan id</help>

<fields>

<field name="">

<mib-index name="vlan-id"/>

</field>

</fields>

</res-id>

<parameter name="name" parameter-type="Vlan::AdminString">

<help>name</help>

<optional default="&quot;&quot;"/>

<fields>

<field name="">

<mib-var tree-node="NODEdot1qVlanStaticName" table-name="dot1qVlanStaticTable"/>

</field>

</fields>

</parameter>

</parameters>

<fields>

<field name="status" basic-type="RowStatus"

access="CommandFieldDefinition::rowStatusAccess\_c">

<mib-var tree-node="NODEdot1qVlanStaticRowStatus"

table-name="dot1qVlanStaticTable"/>

</field>

</fields>

</node>

</nodes>

Properties :

* The attribute is-dynamic of the node is true
* The list of commands is the same as for a static node
* The node contains an auxiliary field with basic-type RowStatus
* The access of the auxiliary field is rowStatusAccess which is a predefined combination of access-rights.
* There is no parameter corresponding to the **dot1qVlanStaticRowStatus** MIB column because this column is already handled by the auxiliary field.
* There is no need for an additional field to verify if the node exists like in a static node. This function is taken over by the auxiliary field.
  + - 1. Variant : create-only parameters

Some parameters may only be entered during creation. In the node XML file we find then :

<parameter name="mode" parameter-type="Vlan::SystemMode" not-entered="modify">

<help>mode</help>

<fields>

<field name="">

<mib-var tree-node="NODEextendVlanStaticSystemMode"

table-name="extendVlanStaticTable"/>

</field>

</fields>

</parameter>

Properties :

* The attribute not-entered with the value modify indicates that this parameter may not be entered by the operator during modify. The operator can thus only enter it during create.
  + - 1. Variant : createAndWait

Some SNMP mappers only allow to modify a table when the instance is not in service. This is typically the case for LANX SNMP mappers.

The node XML looks then like :

<?xml version="1.0" encoding="UTF-8"?>

<nodes xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="file:///cm4/tools/BELL/xml/xmlCommon/node.xsd"

language="CLI">

<node name="tcp" access="AccessRights::all\_rw" is-dynamic="true">

<help>configure tcp filter rules</help>

<man-page file="./CLI\_System/manIssL3FilterTCP.xml"/>

<default-command>

<command handler-type="configure" class="CommandDefinition"/>

</default-command>

<commands>

<command handler-type="info" class="CommandDefinition"/>

<command handler-type="help" class="CommandDefinition"/>

<command handler-type="tree" class="CommandDefinition"/>

</commands>

<parameters>

<res-id uname="index-num" parameter-type="Sys::Number">

<help>L3 Filter rule number</help>

<fields>

<field name="">

<mib-index name="issL3FilterNo"/>

</field>

</fields>

</res-id>

<parameter uname="action-drop" parameter-type="Sys::L3Action">

<help>specifies the action to be taken on the packet</help>

<optional default="&quot;action-allow&quot;"/>

<fields>

<field name="" access="CommandFieldDefinition::mayBeWrittenDuringModify\_c

|CommandFieldDefinition::mayBeReadDuringDisplay\_c">

<mib-var tree-node="NODEissL3FilterAction" table-name="issL3FilterTable" lanx="true"/>

</field>

</fields>

</parameter>

<parameter name="dst-ip-addr" parameter-type=" Ip::V4Address ">

<help>destination IP address to be matched with the packet</help>

<optional default="&quot;0.0.0.0/0&quot;"/>

<fields>

<field name="" access="CommandFieldDefinition::mayBeWrittenDuringModify\_c

|CommandFieldDefinition::mayBeReadDuringDisplay\_c">

<mib-var tree-node="NODEissL3FilterDstIpAddr" table-name="issL3FilterTable"

lanx="true"/>

</field>

</fields>

</parameter>

<display-ele name="row-status" parameter-type="WaitRowStatus">

<help>the status of the instance</help>

<info/>

<fields>

<field name="">

<mib-var tree-node="NODEissL3FilterStatus" table-name="issL3FilterTable"

lanx="true"/>

</field>

</fields>

</display-ele>

</parameters>

<fields>

<field name="status" basic-type="WaitRowStatus"

access="CommandFieldDefinition::waitRowStatusAccess\_c">

<mib-var tree-node="NODEissL3FilterStatus" table-name="issL3FilterTable" lanx="true"/>

</field>

</fields>

</node>

</nodes>

Properties :

* The access attribute for each parameter field corresponding to a column in a MIB table must be redefined as shown. This is necessary because the default access right for a parameter field (see section 2.2.2) contains mayBeWrittenDuringCreate\_c which is typically not allowed by the SNMP mapper.
* The basic-type of the status field is now WaitRowStatus
* The access rights of the status field is now waitRowStatusAccess\_c
* A display element is added to show the status. This allows the operator to verify if the entry is active or not when he executes info detail.
  + 1. Intermediate node
       1. When to use ?

Use this pattern always when a node has **no res-id**, **parameters** or **display-ele**, but only sub-nodes.

* + - 1. Basic construction

The node XML file looks like :

<?xml version="1.0" encoding="UTF-8" ?>

<nodes xmlns:xsi=<http://www.w3.org/2001/XMLSchema-instance>

xsi:noNamespaceSchemaLocation=[file:///cm4/tools/BELL/xml/xmlCommon/node.xsd](file:///F:\cm4\tools\BELL\xml\xmlCommon\node.xsd)

language="CLI">

<node name="atm" access="AccessRights::atm\_rw">

<help>configure ATM</help>

<man-page file="CLI\_Language/manEmpty.xml"/>

<default-command>

<command handler-type="in" class="CommandDefinition"/>

</default-command>

<commands>

<command handler-type="info" class="CommandDefinition"/>

<command handler-type="help" class="CommandDefinition"/>

<command handler-type="tree" class="CommandDefinition"/>

</commands>

<subnode file="CLI\_ATM/atmVclTable.xml"/>

<subnode file="CLI\_ATM/atmQosTable.xml"/>

</node>

</nodes>

Properties :

* The node does not have its own man-page but it uses a fixed one
* The default command is in.
* The sub-nodes are added via the subnode element.

Node : all nodes, except the ones with default command **show** and **poll**, may have subnodes.

* + 1. Admin node
       1. When to use ?

Use this pattern always for an **admin** sub-node in case this node contains a number of commands.

* + - 1. Basic construction

The parameter-type XML file contains :

<parameter-type name="Sys::PrepareShutdown">

<help>prepare the system for shutdown</help>

<fields>

<alternative-field name="" basic-type="SignedInteger">

<alternatives>

<alternative identifier="prepare-shutdown" help="prepare the system for shutdown"

value="(long)1"/>

</alternatives>

</alternative-field>

</fields>

</parameter-type>

<parameter-type name="Sys::RebootShub" >

<help>reboot the shub</help>

<fields>

<alternative-field name="" basic-type="SignedInteger">

<alternatives>

<alternative identifier="reboot-shub" help="reboot the shub"

value="(long)E\_TruthValue\_true"/>

</alternatives>

</alternative-field>

</fields>

</parameter-type>

<parameter-type name="Sys::IsamRestart">

<help>specifies how the isam must be rebooted</help>

<fields>

<alternative-field name="" basic-type="SignedInteger">

<alternatives>

<alternative identifier="with-self-test" help="restart with selftest"

value="(long)E\_EqptAsamRestart\_restartWithSelfTest"/>

<alternative identifier="without-self-test" help="restart without selftest"

value="(long)E\_EqptAsamRestart\_restartWithoutSelfTest"/>

<alternative identifier="default-no-persist" help="restart with all persistent data removed"

value="(long)E\_EqptAsamRestart\_restartFromDefaultPresOAM"/>

<alternative identifier="default-no-data" help="restart with all data removed"

value="(long)E\_EqptAsamRestart\_restartFromDefaultTotal"/>

<alternative identifier="hot-restart" help="reboot, but don't touch the hardware"

value="(long)E\_EqptAsamRestart\_restartHot"/>

<alternative identifier="initial-value" help="no restart - initial value"

value="(long)E\_EqptAsamRestart\_initialValue" access="read"/>

</alternatives>

</alternative-field>

</fields>

</parameter-type>

The node XML file looks like :

<?xml version="1.0" encoding="UTF-8"?>

<nodes xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="file:///cm4/tools/BELL/xml/xmlCommon/node.xsd"

language="CLI">

<node name="equipment" access="AccessRights::equip\_rw">

<help>manage the equipment</help>

<man-page file="CLI\_Equipment/manRestartSystem.xml"/>

<default-command>

<command handler-type="action" class="CommandDefinition"/>

</default-command>

<commands>

<command handler-type="help" class="CommandDefinition"/>

<command handler-type="tree" class="CommandDefinition"/>

</commands>

<parameters>

<parameter uname="prepare-shutdown" parameter-type="Sys::PrepareShutdown">

<help>prepare the system for shutdown</help>

<optional/>

<fields>

<field name="">

<mib-var tree-node="NODE\_asamShutdownBitMap"

table-name="SnmpAgent::localScalarTable\_m"/>

</field>

</fields>

</parameter>

<parameter uname="reboot-shub" parameter-type="Sys::RebootShub">

<help>reboot the shub</help>

<optional/>

<fields>

<field name="">

<mib-var tree-node="NODE\_issRestart" table-name="SnmpAgent::lanxScalarTable\_m"

lanx="true"/>

</field>

</fields>

</parameter>

<parameter name="reboot-isam" parameter-type="Sys::IsamRestart">

<help>reboot the isam (without shub)</help>

<optional/>

<fields>

<field name="">

<mib-var tree-node="NODE\_eqptAsamRestart"

table-name="SnmpAgent::localScalarTable\_m"/>

</field>

</fields>

</parameter>

</parameters>

</node>

</nodes>

The man-page XML contains :

<?xml version="1.0" encoding="UTF-8"?>

<manpage xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="file:///cm4/tools/BELL/xml/xmlCommon/manpage.xsd"

language="CLI">

<userguide>

<title>Alcatel iSAM CLI User Guide</title>

<author>Kumar Shanmugam</author>

<chapter title="Equipment **Management Command**">

<chapter ref="cmd\_descr">

<p>This command allows the operator to manage the equipment.</p>

</chapter>

<chapter ref="usr\_level"/>

<chapter isCommand="true"/>

</chapter>

</userguide>

</manpage>

Properties :

* All commands are modelled as parameters.
* Commands that need no parameters are modelled as unnamed parameters. The parameter-type of an unnamed parameter contains one single alternative with as identifier the name of the command.
* Commands that need parameters are modelled as named parameters. The corresponding parameter-type contains all required parameters.
* All parameters are optional, without default value
* The name of the parameters reflects that it is a command.
* The default command is action.
* The list of environment commands for a sub-node of the admin top-node is always : **tree** and **help**.
* The help of the node starts always with the string **manage**.
* The titel of the man-page ends with the string **Management Command**
  + 1. Show node
       1. When to use ?

Use this pattern always for leaf-nodes of the **show** top-node.

* + - 1. Basic construction

The node XML file looks like :

<?xml version="1.0" encoding="UTF-8" ?>

<nodes xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="file:///cm4/tools/BELL/xml/xmlCommon/node.xsd"

language="CLI">

<node name="snap-shot" access="AccessRights::alarm\_r">

<help>show the alarm snap shot status</help>

<man-page file="CLI\_AlarmReporting/manshowAlarmSnapShotTable.xml"/>

<default-command>

<command handler-type="show" class="CommandDefinition"/>

</default-command>

<commands>

<command handler-type="help" class="CommandDefinition"/>

<command handler-type="tree" class="CommandDefinition"/>

</commands>

<parameters>

<res-id uname="index" parameter-type="Alarm::Index">

<help>alarm snap shot table index</help>

<fields>

<field name="">

<mib-index name="asamAlarmSnapshotIndex"/>

</field>

</fields>

</res-id>

<display-ele name="type" parameter-type="Alarm::Type">

<help>the alarm type</help>

<show when="always"/>

<fields>

<field name="">

<mib-var tree-node="NODEasamAlarmSnapshotType"

table-name="asamAlarmSnapshotTable"/>

</field>

</fields>

</display-ele>

<display-ele name="status" parameter-type="Alarm::Status">

<help>the alarm status.</help>

<show when="extensive"/>

<fields>

<field name="">

<mib-var tree-node="NODEasamAlarmSnapshotStatus"

table-name="asamAlarmSnapshotTable"/>

</field>

</fields>

</display-ele>

</parameters>

<fields>

<field name=”in" basic-type="SignedInteger"

access="CommandFieldDefinition::mayBeReadDuringIn\_c">

<mib-var tree-node="NODEasamAlarmSnapshotType"

table-name="asamAlarmSnapshotTable"/>

</field>

</fields>

</node>

</node>

The man page XML contains :

<?xml version="1.0" encoding="UTF-8"?>

<manpage xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="file:///cm4/tools/BELL/xml/xmlCommon/manpage.xsd"

language="CLI">

<userguide>

<title>Alcatel iSAM CLI User Guide</title>

<author>Kumar Shanmugam</author>

<chapter title="Alarm Snap-shot **Status Command**">

<chapter ref="cmd\_descr">

<p>This command shows the entries in the alarm snap shot table.</p>

</chapter>

<chapter ref="usr\_level"/>

<chapter isCommand="true"/>

</chapter>

</userguide>

</manpage>

Properties :

* The help of the node starts with the string show and ends with status
* The default command is show
* The list of environment commands for a leaf-node of the show top-node is always : **tree** and **help**.
* The first parameter is always shown.
* The second parameter (extensive) is only shown when the operator selects the **detail** option for the show command. All parameters that are shown when the operator selects no **detail** option must fit in a table of 80 characters wide. Showing parameters only when the operator selects **detail**, can help to archieve this requirement.
* For tables for which the system does not guarantee that an instance exists for all possible combinations of the indices, must a special node field be added to the node.
* This auxiliary field will be read to verify if the node instance exists or not. That is why the access is mayBeReadDuringIn.
* The MIB variable related to the special node field is chosen arbitrary : it can be any of the columns of the MIB table.
* The title of the man page must end with the string **Status Command**
  + - 1. Variant : scalar variables

It is recommended to combine scalar variables with the related table as in following example. It is also possible to use the basic pattern without res-id, but this should only be done when there is no related table.

The node XML file contains :

<?xml version="1.0" encoding="UTF-8" ?>

<nodes xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="file:///cm4/tools/BELL/xml/xmlCommon/node.xsd"

language="CLI">

<node name="snap-shot" access="AccessRights::alarm\_r">

<help>show the alarm snap shot status</help>

<man-page file="CLI\_AlarmReporting/manshowAlarmSnapShotTable.xml"/>

<default-command>

<command handler-type="show" class="CommandDefinition"/>

</default-command>

<commands>

<command handler-type="help" class="CommandDefinition"/>

<command handler-type="tree" class="CommandDefinition"/>

</commands>

<parameters>

<res-id uname="index" parameter-type="Alarm::Index">

<help>alarm snap shot table index</help>

<fields>

<field name="">

<mib-index name="asamAlarmSnapshotIndex"/>

</field>

</fields>

</res-id>

<display-ele name="type" parameter-type="Alarm::Type">

<help>the alarm type</help>

<show when="always"/>

<fields>

<field name="">

<mib-var tree-node="NODEasamAlarmSnapshotType"

table-name="asamAlarmSnapshotTable"/>

</field>

</fields>

</display-ele>

<display-ele name="status" parameter-type="Alarm::Status">

<help>the alarm status.</help>

<show when="extensive"/>

<fields>

<field name="">

<mib-var tree-node="NODEasamAlarmSnapshotStatus"

table-name="asamAlarmSnapshotTable"/>

</field>

</fields>

</display-ele>

</parameters>

<sections>

<section name="common" is-common="true">

<help>the status common for all snapshot entries</help>

<display-ele name="reservation-period" parameter-type="TimeTicks">

<help>the remaining time that the snapshot table is reserved</help>

<show when="always"/>

<fields>

<field name="">

<mib-var tree-node="NODE\_asamAlarmSnapshotTimeout"

table-name="SnmpAgent::localScalarTable\_m"/>

</field>

</fields>

</display-ele>

<display-ele name="entries" parameter-type="Gauge">

<help>the number of entries in the snapshot table</help>

<show when="always"/>

<fields>

<field name="">

<mib-var tree-node="NODE\_asamNumberOfAlarmSnapshotEntries"

table-name="SnmpAgent::localScalarTable\_m"/>

</field>

</fields>

</display-ele>

<section>

</sections>

<fields>

<field name=”in" basic-type="SignedInteger"

access="CommandFieldDefinition::mayBeReadDuringIn\_c">

<mib-var tree-node="NODEasamAlarmSnapshotType"

table-name="asamAlarmSnapshotTable"/>

</field>

</fields>

</node>

</node>

Properties :

* A common section is introduced
* The attribute is-common of the section is true
* The table-name attribute contains SnmpAgent::localScalarTable\_m or SnmpAgent::lanxScalarTable\_m
  + - 1. Sections

Sections are used when the number of **display-ele** becomes very large and one wants to group the **display-ele** together.

The node XML file becomes :

<?xml version="1.0" encoding="UTF-8"?>

<nodes xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="file:///cm4/tools/BELL/xml/xmlCommon/node.xsd"

language="CLI">

<node name="vrf-statistics" access="AccessRights::ip\_r">

<help>show the VRF statistics status.</help>

<man-page file="./CLI\_IP/manshowvrfStatistics.xml"/>

<default-command>

<command handler-type="show" class="CommandDefinition"/>

</default-command>

<commands>

<command handler-type="help" class="CommandDefinition"/>

<command handler-type="tree" class="CommandDefinition"/>

</commands>

<parameters>

<res-id uname="vrf-id" parameter-type="AsamProfileIndex">

<help>an unique id of a VRF.</help>

<fields>

<field name="">

<mib-index name="vrfIndex"/>

</field>

</fields>

</res-id>

<display-ele name="ip-route-entries" parameter-type="Counter">

<help>the total number of routes in the system.</help>

<show when="always"/>

<fields>

<field name="">

<mib-var tree-node="NODEvrfStatsCurrentIpRouteEntries" table-name="vrfStatsTable"/>

</field>

</fields>

</display-ele>

<display-ele name="ip-net2media-entries" parameter-type="Counter">

<help>the total number net2media entries.</help>

<show when="always"/>

<fields>

<field name="">

<mib-var tree-node="NODEvrfStatsCurrentIpNetToMediaEntries"

table-name="vrfStatsTable"/>

</field>

</fields>

</display-ele>

</parameters>

<sections>

<section name="system statistics">

<help>show the VRF system statistics status.</help>

<display-ele name="ip-route-entries" parameter-type="Counter">

<help>the total number of IP routes in the system.</help>

<show when="extensive"/>

<fields>

<field name="">

<mib-var tree-node="NODEvrfSysStatsCurrentIpRouteEntries"

table-name="vrfSysStatsTable"/>

</field>

</fields>

</display-ele>

<display-ele name="ip-net2media-entries" parameter-type="Counter">

<help>the total number of IP net2media entries.</help>

<show when="extensive"/>

<fields>

<field name="">

<mib-var tree-node="NODEvrfSysStatsCurrentIpNetToMediaEntries"

table-name="vrfSysStatsTable"/>

</field>

</fields>

</display-ele>

</section>

</sections>

<fields>

<field name="exists" basic-type="SignedInteger"

access="CommandFieldDefinition::mayBeReadDuringIn\_c">

<mib-var tree-node="NODEvrfStatsCurrentIpRouteEntries" table-name="vrfStatsTable"/>

</field>

</fields>

</node>

</nodes>

Properties :

* A section is added
* A section is only shown when the operator specifies the **detail** option with the show command.
* The name of the section is in lower case and may contain spaces.
  + - 1. Variant : short names

The default format for show is a table format. The XML file must be designed in such a way that each table row can be displayed on a screen of 80 characters wide. This can be realised in following ways:

* Show some parameters only in the detail view (<show when=”extensive”/>
* Reduce the number of characters needed to display a parameter

The parameter-type XML file contains :

<parameter-type name="Xdsl::SignalStatus">

<help>the signal status on the xDSL line</help>

<fields>

<alternative-field name="" basic-type="SignedInteger">

<alternatives>

<alternative identifier="lost" short-name=”yes” help="no signal present"

value="(long)1"/>

<alternative identifier="present" short-name=”no” help="signal present"

value="(long)2"/>

</alternatives>

</alternative-field>

</fields>

</parameter-type>

The node XML file contains :

<display-ele name="signal" short-name=”los” parameter-type="Alarm::Status">

<help>the signal status on the xDSL line</help>

<show when="always"/>

<fields>

<field name="">

<mib-var tree-node="NODExdslNearEndLossOfSignal"

table-name="asamXdslNearEndTable"/>

</field>

</fields>

</display-ele>

The column in the table will now be 3 characters wide instead of 7 in case no short-name tags would be used. The title will be los (a well known abbreviation of loss of signal) and the values yes or no.

The detail mode will be signal : present or signal : lost.

The default value in for short-name is the value of name or identifier. So it is not necessary to define at the same time short-name for the parameter name and the enumeration fields.

* + - 1. Variant : split header line

Sometimes it is useful to place a common header on top of two or more adjacent columns.

This can be done by putting in the short-name a string in which the common header is separated from the column title with a slash ‘**/**’.

Example:

**<?xml version="1.0" encoding="UTF-8" ?>**

**<nodes xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"**

**xsi:noNamespaceSchemaLocation="file:///cm4/tools/BELL/xml/xmlCommon/node.xsd"**

**language="CLI">**

**<node name="manager" access="AccessRights::transport\_r">**

**<help>show the statistics for an SNMP manager</help>**

**<man-page file="CLI\_Transport/manshowSnmpStatistics.xml"/>**

**<default-command>**

**<command handler-type="show" class="CommandDefinition"/>**

**</default-command>**

**<commands>**

**<command handler-type="help" class="CommandDefinition"/>**

**<command handler-type="tree" class="CommandDefinition"/>**

**</commands>**

**<parameters>**

**<res-id uname=”manager" parameter-type="Ip::UdpPort">**

**<help>the address of the manager</help>**

**<fields>**

**<field name="">**

**<mib-index name="manager"/>**

**</field>**

**</fields>**

**</res-id>**

**<display-ele name="rx-get" short-name=”received/gets” parameter-type="Gauge">**

**<help>the number of get-requests received</help>**

**<show when="always"/>**

**<fields>**

**<field name="">**

**<mib-var tree-node="NODEsnmpManagerGetRequests"**

**table-name="snmpManagerTable"/>**

**</field>**

**</fields>**

**</display-ele>**

**<display-ele name="rx-getnext" short-name=”received/get-nexts” parameter-type="Gauge">**

**<help>the number of getnext-requests received</help>**

**<show when="always"/>**

**<fields>**

**<field name="">**

**<mib-var tree-node="NODEsnmpManagerGetNextRequests"**

**table-name="snmpManagerTable"/>**

**</field>**

**</fields>**

**</display-ele>**

**<display-ele name="rx-set" short-name=”received/sets” parameter-type="Gauge">**

**<help>the number of set-requests received</help>**

**<show when="always"/>**

**<fields>**

**<field name="">**

**<mib-var tree-node="NODEsnmpManagerGetNextRequests"**

**table-name="snmpManagerTable"/>**

**</field>**

**</fields>**

**</display-ele>**

**<display-ele name="tx-resp" short-name=”transmitted/responses” parameter-type="Gauge">**

**<help>the number of responses transmitted</help>**

**<show when="always"/>**

**<fields>**

**<field name="">**

**<mib-var tree-node="NODEsnmpManagerResponses"**

**table-name="snmpManagerTable"/>**

**</field>**

**</fields>**

**</display-ele>**

**</parameters>**

**</node>**

**</node>**

This will result in following table header:

**=============================================================**

**| received |transmitted**

**manager |gets get-nexts sets |responses**

**---------------+----------+----------+----------+------------**

* + - 1. Variant : combined configuration and show

A good example is the traceroute. The implementation requires the following steps :

* Create an entry in a MIB table
* Read all the rows in a table
* Delete the entry in the MIB table

Solution :

* Create a normal show command (see section 5.2.6)
* Replace the default command **show** by **sample**.
* Add the necessary **parameter** elements and node-specific auxiliary fields like in a **configure** command (see sections 5.2.1, 5.2.2, 5.2.3 …).
  + - 1. Variant : combined configuration and show with fixed delay

Sometimes it is necessary to wait a fixed amount of time between the **commit** phase and the **display** phase.

Solution :

* Create an auxiliary field with name **sample-delay** which may exists in the context during the **commit** phase and that has as value the required delay in milliseconds.

We will wait 2 seconds if the node XML file contains :

<fields>

<field name="sample-delay" basic-type="SignedInteger"

access="CommandFieldDefinition::mayBeUsedDuringCommit\_c"

default=”2000L”>

</field>

</fields>

* + - 1. Variant : filters

Sometimes we want to show only the entries of a particular kind and contains the MIB table entries of different kinds.

The **show** command will already filter out those entries for which one of the read field values is not valid according to the corresponding basic-type. But this is sometimes not sufficient.

To show only the entries that have a given value (in our example 7) we could use following construct in the node XML file :

<fields>

<field name=”filter” basic-type=”SignedInteger”

access=”CommandFieldDefinition::mayBeReadDuringDisplay\_c

|CommandFieldDefinition::isFilter\_c”

default=”7L”>

<mib-var tree-node="NODE\_asamIfType"

table-name="asamIfTable"/>

</field>

</fields>

Solution :

* An auxiliary field is added to the node.
* The auxiliary field is readable during display and is marked as a filter. This last property will make that the auxiliary field will be read even if it already has a default value.
* The auxiliary field has as default value with the value that valid instances have.
* The auxiliary field is linked to the MIB variable.
  + - 1. Variant : only detail output

For some show commands it does not make sense to provide a table representation because the number of data to be shown is too large. In this case can the detail output be enforced.

Solution:

* Add the option CliShowCommand::onlyDetail\_c to the show handler.

Example:

<default-command>

<command handler-type="show" class="CommandDefinition"

options=”((signed long)CliShowCommand::onlyDetail\_c)”/>

</default-command>

1. pATTERNS FOR EXPERIENCED CLI DESIGNERS
   1. Type patterns
      1. Bit masks
         1. When to use ?

Use this pattern when bits or group of bits in a signed integer have each a specific meaning and the operator must be able to modify and or display individual fields.

Don’t use this pattern when a signed integer must be interpreted as a list of objects (ports, alarms, ….). Use then the patterns explained in sections 6.2.1 and 6.2.2

* + - 1. Basic construction

In the following example contains the integer 3 groups of bits : 2 groups (**atm** and **aaa**) that can be interpreted, 1 group that contains a small number (**priority**).

The CLI\_Agent/AccessRights.hpp header file contains :

class AccessRights {

public :

typedef enum AccessGroup {

aaa\_r = 1<<0,

aaa\_w = 2<<0,

aaa\_rw = 3<<0,

atm\_r = 1<<2,

atm\_w = 2<<2,

atm\_rw = 3<<2

priority\_mask = 7<<4,

all\_r = 1<<7,

all\_w = 2<<7,

all\_rw = 3<<7

} AccessGroup;

The basic-types XML file contains :

<include file="CLI\_Agent/AccessRights.hpp"/>

<basic-type name="Sec::Aaa" class="SingleMaskType">

<help>specifies the access rights to the AAA related commands</help>

<option>AccessRights::aaa\_rw</option>

</basic-type>

<basic-type name="Sec::Atm" class="SingleMaskType">

<help>specifies the access rights to the ATM related commands</help>

<option>AccessRights::atm\_rw</option>

</basic-type>

The parameter-types XML file contains :

<parameter-type name="Sec::Aaa">

<help>specifies the access rights to the AAA related commands</help>

<fields>

<alternative-field name="" basic-type="Sec::Aaa">

<alternatives>

<alternative identifier="write" help="write and read access"

value="(long)AccessRights::aaa\_rw"/>

<alternative identifier="read" help="read access" value="(long)AccessRights::aaa\_r"/>

<alternative identifier="none" help="no access" value="(long)0"/>

</alternatives>

</alternative-field>

</fields>

</parameter-type>

<parameter-type name="Sec::Atm">

<help>specifies the access rights to the ATM related commands</help>

<fields>

<alternative-field name="" basic-type="Sec::Atm">

<alternatives>

<alternative identifier="write" help="write and read access"

value="(long)AccessRights::atm\_rw"/>

<alternative identifier="read" help="read access" value="(long)AccessRights::atm\_r"/>

<alternative identifier="none" help="no access" value="(long)0"/>

</alternatives>

</alternative-field>

</fields>

</parameter-type>

The node XML file contains :

<?xml version="1.0" encoding="UTF-8"?>

<nodes xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="file:///cm4/tools/BELL/xml/xmlCommon/node.xsd"

language="CLI">

<node name="profile" access=”AccessRights::forAdmin\_c" is-dynamic="true">

<help>configure the operator profile</help>

<man-page file="./CLI\_Operators/manConfigProfile.xml"/>

<default-command>

<command handler-type="configure" class="CommandDefinition"/>

</default-command>

<commands>

<command handler-type="info" class="CommandDefinition"/>

<command handler-type="help" class="CommandDefinition"/>

<command handler-type="tree" class="CommandDefinition"/>

</commands>

<parameters>

<res-id uname="name" parameter-type="Sec::ProfileName">

<help>the name of the profile</help>

<fields>

<field name="" access="CommandFieldDefinition::mayAlwaysBeAccessedViaNode\_c"/>

</fields>

</res-id>

<parameter name="aaa" parameter-type="Sec::Aaa">

<help>the access rights to AAA related commands</help>

<optional default="&quot;none&quot;"/>

<fields>

<field name="">

<master name="\_accessr"/>

</field>

</fields>

</parameter>

<parameter name="atm" parameter-type="Sec::Atm">

<help>the access rights to ATM related commands</help>

<optional default="&quot;none&quot;"/>

<fields>

<field name="">

<master name="\_accessr"/>

</field>

</fields>

</parameter>

</parameters>

<fields>

<field name="accessr" basic-type="CombinedMaskType"

access="CommandFieldDefinition::mayBeReadDuringDisplay\_c

|CommandFieldDefinition::mayBeWrittenDuringCreate\_c

|CommandFieldDefinition::mayBeWrittenDuringModify\_c">

<mib-var tree-node="NODEoperatorProfileAccess" table-name="operatorProfileTable"/>

</field>

<field name="status" basic-type=”RowStatus"

access="CommandFieldDefinition::rowStatusAccess\_c”>

<mib-var tree-node="NODEoperatorProfileStatus" table-name="operatorProfileTable"/>

</field>

</fields>

</node>

</nodes>

Properties :

* A basic type is created for each bit or group of bits.
* The class of such basic type must be **SingleMaskType**.
* The option of such basic type must be the mask. This is an integer in which all bits related to the group are set and all other bits are cleared. The value of the mask is in this example defined by a C-enumeration value.
* A parameter type is created for each bit or group of bits. In the current example has each parameter type one alternative-field limiting the possible.
* A parameter is added in the node XML for each group of bits
* An auxiliary field accessr is added that is linked to the mib variable.
* The field of each bit group must have a master element. The **name** attribute of this master element must have as value the full name of a field. The full name of a field is <*parameter-name*>**\_**<*field-name*>. The full name of an auxiliary field is \_<*field-name*>. The full name of our auxiliary field accessr is \_accessr.
* The value of the auxiliary field accessr must be written during create and modify. It must be read during display.

The CombinedMaskType class will do the following to calculate the integer value to be sent to SNMP :

* in case of modify : read the current integer value else initialise the integer value to 0
* loop over each field of which the master attribute contains the full name of the CombinedMaskType field.
  + if the slave field has a value : replace in the integer value the value of the bit-group with the new value
* assign the integer value in case of create if all slave fields had a value or in case of modify.

The CombinedMaskType class will do the following when the value is read via SNMP :

* loop over each field of which the master attribute contains the full name of the CombinedMaskType field.
  + assign the value of the bit-group to the slave field.
    - 1. Variant : a numeric value is entered for a bit-field

Following lines are added to the basic-types XML file :

<basic-type name=”Sec::Priority” class=”SingleMaskType”>

<help>the priority for operators belonging to this profile</help>

<option>7, 4</option>

<restrictions>

<range min-included=”0” max-included=”7” action=”accept” type=”signed”/>

</restrictions>

</basic-type>

Following lines are added to the parameter-types XML file :

<parameter-type name="Sec::Priority">

<help>specifies the priority for operators belonging to this profile</help>

<fields>

<field name="" basic-type="Sec::Priority"/>

</fields>

</parameter-type>

Following lines are added to the node XML file :

<parameter name="priority" parameter-type="Sec::Priority">

<help>the priority for operators belonging to this profile</help>

<optional default="1L"/>

<fields>

<field name="">

<master name="\_accessr"/>

</field>

</fields>

</parameter>

Properties :

* A basic type is created for each group of bits.
* The class of such basic type must be **SingleMaskType**.
* The first option of such basic type must be the relative mask. This is an integer in which the lowest N bits are set and all other bits are cleared. N is the number of bits in the bit group. N is in our case equal to 3.
* The second option to the basic type indicates how many bits the value entered by the operator must be shifted to the left. This value is in our case equal to 4.
* The possible values of the basic type must be limited to a range of which all values fit into the mask.
* A parameter type is created for each bit or group of bits.
* A parameter is added in the node XML for each group of bits
* The field of each bit group must have a master element. The **name** attribute of this master element must have as value the full name of the auxiliary field.
  + - 1. Variant : separate SNMP variables to read, set and clear

The MIB contains 3 variables 1 to read the status, 1 to clear bits and 1 to set bits :

futOspfRRDSrcProtoMaskEnable OBJECT-TYPE

SYNTAX INTEGER

ACCESS read-write

STATUS mandatory

DESCRIPTION

"The bit mask of source protocols from which routes are

imported to OSPF.

Bit 1, if set, indicates that local routes are

redistributed into OSPF. (value = 2)

Bit 7, if set, indicates that routes from RIP are

redistributed into OSPF. (value = 128) "

DEFVAL { 0 }

::= { futospfRRDGeneralGroup 2 }

futOspfRRDSrcProtoMaskDisable OBJECT-TYPE

SYNTAX INTEGER

ACCESS read-write

STATUS mandatory

DESCRIPTION

"The bit mask of source protocols from which redistribution

to OSPF is disabled.

Bit 1, if set, indicates that local routes are not

redistributed into OSPF. (value = 2)

Bit 7, if set, indicates that routes from RIP are not

redistributed into OSPF. (value = 128)”

DEFVAL { 0 }

::= { futospfRRDGeneralGroup 3 }

futOspfRRDSrcProtoStatus OBJECT-TYPE

SYNTAX INTEGER

ACCESS read-only

STATUS mandatory

DESCRIPTION

"The bit mask of source protocols from which redistribution

to OSPF actually is enabled.

Bit 1, if set, indicates that local routes are not

redistributed into OSPF. (value = 2)

Bit 7, if set, indicates that routes from RIP are not

redistributed into OSPF. (value = 128)”

DEFVAL { 0 }

::= { futospfRRDGeneralGroup 4 }

Following lines must be added to the basic-types XML file :

<basic-type name="Ospf::SetClear" class="CombinedMaskType">

<help>manages a bit mask via set and clear fields</help>

<option>&quot;\_set&quot;,&quot;\_clear&quot;</option>

</basic-type>

The definition of the accessr field in the node XML file must be replaced by following lines :

<field name="accessr" basic-type="Ospf::SetClear"

access="CommandFieldDefinition::mayBeUsedDuringCreate\_c

|CommandFieldDefinition::mayBeUsedDuringModify\_c

|CommandFieldDefinition::mayBeReadDuringDisplay\_c">

<mib-var tree-node="NODE\_futOspfRRDSrcProtoStatus"

table-name="SnmpAgent::lanxScalarTable\_m" lanx="true"/>

</field>

<field name="set" basic-type="SignedInteger"

access="CommandFieldDefinition::mayBeWrittenDuringCreate\_c

|CommandFieldDefinition::mayBeWrittenDuringModify\_c">

<mib-var tree-node="NODE\_futOspfRRDSrcProtoMaskEnable"

table-name="SnmpAgent::lanxScalarTable\_m" lanx="true"/>

</field>

<field name="clear" basic-type="SignedInteger"

access="CommandFieldDefinition::mayBeWrittenDuringCreate\_c

|CommandFieldDefinition::mayBeWrittenDuringModify\_c">

<mib-var tree-node="NODE\_futOspfRRDSrcProtoMaskDisable"

table-name="SnmpAgent::lanxScalarTable\_m" lanx="true"/>

</field>

Properties :

* Instead of one auxiliary field accessr we have now three fields : accessr, set and clear.
* The accessr field is linked to the MIB variable that must be used for reading.
* A special basic-type is created for this field : Ospf::SetClear.
* This basic-type has as class CombinedMaskType.
* The first option of the basic-type is the full name of the field to set the bits (in this case the auxiliary field set, so the full name is \_set).
* The second option of the basic-type is the full name fo the field to clear the bits (in this case the auxiliary field clear, so the full name is \_clear).
* The accessr field must be used during create and modify and read during display.
* The set field is linked to the MIB variable that must be used to set bits.
* The clear field is linked to the MIB variable that must be used to clear bits.
* The basic type of the set and clear fields is SignedInteger.
* The set and clear fields must be written during create and modify.
  + - 1. Variant : a number of bits must be set to a fixed value

In this example we want that a number of bits (AccessRights::all\_rw) are set when the instance is created.

Following lines must be added to the basic-type XML file :

<basic-type name="Sec::All" class="SingleMaskType">

<help>specifies the default rights for an operator</help>

<option>AccessRights::all\_rw</option>

</basic-type>

Following lines must then be added in the **fields** section of the node XML file :

<field name="default" basic-type="Sec::All"

access="CommandFieldDefinition::mayBeUsedDuringCreate\_c"

default="(long)AccessRights::all\_rw">

<master name="\_accessr"/>

</field>

Properties :

* A basic type is created as explained in section 6.1.1.2.
* An additional auxiliary field default is added to the node.
* The auxiliary field has as default the bits to be set.
* The field may be written during create (because only then we have to add the bits to the mask).
* The master of the auxiliary field accessr.
  + - 1. Variant : binary string

We can also work with bits in a binary string. The actual implementation only allows to work with individual bits.

The basic-types XML file contains :

<basic-type name="Bitmask\_0" class="BitFieldType">

<help>bit mask</help>

<option>0</option>

</basic-type>

<basic-type name="Bitmask\_1" class="BitFieldType">

<help>bit mask</help>

<option>1</option>

</basic-type>

<basic-type name="Vlan::BitMaskType" class="BitMaskType">

<help>bit mask</help>

<length min="1" max="1"/>

</basic-type>

The parameter-types XML file contains :

<parameter-type name="Vlan::BestEffort">

<help>controls the best effort priority</help>

<fields>

<alternative-field name="" basic-type="Bitmask\_0">

<alternatives>

<alternative identifier="no-prior-best-effort" help="disable best effort priority"

value="0L"/>

<alternative identifier="prior-best-effort" help="enable best effort priority"

value="1L"/>

</alternatives>

</alternative-field>

</fields>

</parameter-type>

<parameter-type name="Vlan::Background">

<help>controls the background priority</help>

<fields>

<alternative-field name="" basic-type="Bitmask\_1">

<alternatives>

<alternative identifier="no-prior-background" help="disable background priority"

value=”0L"/>

<alternative identifier="prior-background" help="enable background priority"

value="1L”/>

</alternatives>

</alternative-field>

</fields>

</parameter-type>

The node XML file contains :

<node name="vlan-id" access="(AccessRights::AccessGroup)(AccessRights::vlan\_rw)"

is-dynamic="true" >

<help>configure association of bridge port to vlan</help>

<man-page file="./CLI\_VLAN/manConfigurePortVlanId.xml"/>

<default-command>

<command handler-type="configure" class="CommandDefinition"/>

</default-command>

<commands>

<command handler-type="info" class="CommandDefinition"/>

<command handler-type="help" class="CommandDefinition"/>

<command handler-type="tree" class="CommandDefinition"/>

</commands>

<parameters>

<res-id uname="index" parameter-type="Vlan::StackedVlan">

<help>vlan id</help>

<fields>

<field name="">

<mib-index name="dot1qVlanIndex"/>

</field>

</fields>

</res-id>

<parameter uname="prior-best-effort" parameter-type="Vlan::BestEffort">

<help>enable best effort priority</help>

<optional default="&quot;no-prior-best-effort&quot;"/>

<fields>

<field name="">

<master name="\_pbitMaster"/>

</field>

</fields>

</parameter>

<parameter uname="prior-background" parameter-type="Vlan::Background">

<help>enable background priority</help>

<optional default="&quot;no-prior-background&quot;"/>

<fields>

<field name="">

<master name="\_pbitMaster"/>

</field>

</fields>

</parameter>

</parameters>

<fields>

<field name="extendPortVlanRowStatus" basic-type="RowStatus"

access="CommandFieldDefinition::rowStatusAccess\_c">

<mib-var tree-node="NODEextendPortVlanRowStatus"

table-name="extendPortVlanAssociationTable"/>

</field>

<field name="pbitMaster" basic-type="Vlan::BitMaskType"

access="CommandFieldDefinition::mayBeWrittenDuringCreate\_c|

CommandFieldDefinition::mayBeWrittenDuringModify\_c|

CommandFieldDefinition::mayBeReadDuringDisplay\_c">

<mib-var tree-node="NODEextendPortVlanPbitList"

table-name="extendPortVlanAssociationTable"/>

</field>

</fields>

</node>

</nodes>

Properties :

* A special basic-type is created for the binary string.
* The class of the basic-type is BitMaskType.
* The length of the basic-type indicates how many bytes there are in the binary string. In our case the length is 1.
* A basic-type Bitmask\_*X* is created for each relevant bit (*X* is the number of the bit). These basic-types could be easily reused by other applications.
* The class of the bitmask basic-type is BitFieldType.
* The option of the bitmask basic-type is the number *X* of the bit. The least significant bit of the first byte has number 0. In this case we have made bitmask basic-types for the two least significant bits of the first byte.
* For each bit a parameter-type is created, with a single alternative-field.
* The value of the different alternatives is always 0 or 1 : 0 to clear the bit, 1 to set the bit.
* The binary string is an auxiliary field of the node.
* The master of all the fields corresponding to bits in the binary string is the full name of the auxiliary field.
  + - 1. Variant : binary string separate variables to set and clear

Following lines must be added to the basic-types XML file :

<basic-type name="Vlan::BitMaskType" class="BitMaskType">

<help>a bit mask with bits to set and clear via special fields</help>

<length min="1" max="1"/>

<option>&quot;\_set&quot;,&quot;\_clear&quot;</option>

</basic-type>

<basic-type name="Vlan::ClearSetType" class="BinaryStringType">

<help>a binary string to set and clear bits</help>

<length min="1" max="1"/>

</basic-type>

The definition of the pBitMaster field in the node XML file must be replaced by following lines :

<field name="pBitMaster" basic-type="Vlan::BitMaskType"

access="CommandFieldDefinition::mayBeUsedDuringCreate\_c

|CommandFieldDefinition::mayBeUsedDuringModify\_c

|CommandFieldDefinition::mayBeReadDuringDisplay\_c">

<mib-var tree-node="NODEextendPortVlanPbitList"

table-name="extendPortVlanAssociationTable"/>

</field>

<field name="set" basic-type="Vlan::ClearSetType"

access="CommandFieldDefinition::mayBeWrittenDuringCreate\_c

|CommandFieldDefinition::mayBeWrittenDuringModify\_c">

<mib-var tree-node="NODEextendPortVlanPbitSet"

table-name="extendPortVlanAssociationTable"/>

</field>

<field name="clear" basic-type="Vlan::ClearSetType"

access="CommandFieldDefinition::mayBeWrittenDuringCreate\_c

|CommandFieldDefinition::mayBeWrittenDuringModify\_c">

<mib-var tree-node="NODEextendPortVlanPbitClear"

table-name="extendPortVlanAssociationTable"/>

</field>

Properties :

* The names of the fields to clear and set bits are added as an option to the basic-type Vlan::BitMaskType.
* A separate basic type that is a binary string of the length 1 (must be identical to the length of Vlan::BitMaskType) is created for the set and clear fields
  + 1. Ignore SNMP errors
       1. When to use ?

Use this pattern only when you know that an SNMP variable will return an error code which is not valid according to SNMP. Normally you must write a fault report to get the mapper corrected, but there are some exceptional cases where this is not possible (for example : because changing the mapper would have a large impact on other applications).

* + - 1. Basic implementation

As example we will use following example :

asamNumberOfAlarmSnapshotEntries OBJECT-TYPE

SYNTAX Gauge

ACCESS read-only

STATUS mandatory

DESCRIPTION

"When the snapshot table status is in the inUse state, this object

reflects the number of entries in the snapshot table. If the snapshot

table status is set to notInUse, this object can not be read. "

::= {asamAlarmMIB 20}

This object is a scalar object, so according to SNMP it should always be readable. The mapper returns however **noSuchName** when we try to do so when the snapshot table is not in use.

The basic-type XML file contains :

<basic-type name="Gauge" class="UnsignedIntegerType">

<help>gauge</help>

<other-implementations>

<basic-type name="IgnoredGauge" class="IgnoredUnsignedIntegerType"/>

</other-implementations>

</basic-type>

The parameter-type XML file contains :

<parameter-type name="IgnoredGauge">

<help>an unsigned integer value</help>

<fields>

<field name="" basic-type="IgnoredGauge"/>

</fields>

</parameter-type>

The node XML file contains :

<display-ele name="entries" parameter-type="IgnoredGauge">

<help>specifies nbr of entries in the snapshot table.</help>

<show when="always"/>

<fields>

<field name="">

<mib-var tree-node="NODE\_asamNumberOfAlarmSnapshotEntries"

table-name="SnmpAgent::localScalarTable\_m"/>

</field>

</fields>

</display-ele>

Properties :

* A new basic type IgnoredGauge is created that is an other implementation of Gauge.
* The class of IgnoredGauge is IgnoredUnsignedInteger while the class of Gauge was UnsignedInteger.
* IgnoredUnsignedInteger is a class derived from UnsignedInteger that defines an additional function that assigns 0 to a field when an SNMP error was detected during an SNMP get operation.

Note : other **Ignored**<*Type* > classes exist.

* + 1. Index combines values of different fields
       1. When to use ?

Use this pattern when the operator has to enter many fields, but where these fields must be combined to one index to be used via SNMP. Please use the new index mechanism as much as you can to implement the code that calculates the index. This is explained in section 8.

* + - 1. Basic implementation

The basic-type XML file contains :

<basic-type name="Vlan::UVlanIndex" class="UnsignedIntegerType">

<help>unstacked vlan identity </help>

<restrictions>

<range min-included="1" max-included="4093" action="accept" type="unsigned"/>

<range min-included="0" max-included="0" action="accept" type="unsigned" hide="true"/>

</restrictions>

</basic-type>

<basic-type name="Vlan::SVlanIndex" class="UnsignedIntegerType">

<help>service vlan identity</help>

<restrictions>

<range min-included="0" max-included="500" action="accept" type="unsigned"/>

</restrictions>

</basic-type>

<basic-type name="Vlan::CVlanIndex" class="UnsignedIntegerType">

<help>customer vlan identy </help>

<restrictions>

<range min-included="0" max-included="4093" action="accept" type="unsigned"/>

</restrictions>

</basic-type><basic-type name="Vlan::StackedVlan" class="StackedVlanType">

<help>Stacked Vlan Combination : S-Vlan:C-Vlan </help>

</basic-type>

The parameter-type XML file contains :

<parameter-type name="Vlan::StackedVlan">

<help>generic vlan identity</help>

<fields>

<alternative-field name="unstacked" basic-type="Vlan::UVlanIndex" separator=":">

<alternatives>

<alternative identifier="stacked" help="stacked vlan identity" value="0L" version="2"/>

<default help="unstack vlan identity" version="1"/>

</alternatives>

</alternative-field>

<field name="svlan" basic-type="Vlan::SVlanIndex" separator=":" version="2"/>

<field name="cvlan" basic-type="Vlan::CVlanIndex" version="2"/>

<field name="" basic-type="Vlan::StackedVlan" version="32"/>

</fields>

</parameter-type>

The node XML file contains :

<parameter name="pvid" parameter-type="Vlan::StackedVlan">

<help>default vlan id for untagged frames</help>

<optional default="(long)4097"/>

<fields>

<field name="" >

<mib-var tree-node="NODEdot1qPvid" table-name="dot1qPortVlanTable"/>

</field>

</fields>

</parameter>

Properties :

* The values of the fields unstacked, svlan and cvlan can be entered by the operator.
* The operator can not give a value to the field with an empty string as the version 32 is not a version that appears for the first field (versions 1 and 2 do). This field is an auxiliary parameter field.
* The unstacked field has a special value 0 to indicate that the operator wants to enter a stacked vlan. This value is marked as hide in the basic-type, so that it will not appear in the range shown in the help text.
* The special class StackedVlan is linked to the basic-type of the auxiliary parameter field. This class has the necessary functions to :
  + read the values entered by the operator in the fields unstacked, svlan and cvlan and to calculate an index from it.
  + give the fields unstacked, svlan and cvlan a value depending on the read index.
* The auxiliary parameter field is the only one that is linked to an SNMP variable. It is recommended that a parameter field linked to an SNMP variable has the empty string as name.
  + - 1. Variant : the index field is not an auxiliary parameter field

Older class implementations reuse one of the fields to be entered by the operator for the index.

The basic-type XML file contains :

<basic-type name="Eqpt::Shelf" class="SignedIntegerType">

<help>the physical number of the shelf within the rack</help>

<restrictions>

<range min-included="1" max-included="2" action="accept" type="signed"/>

</restrictions>

<other-implementations>

<basic-type name="Equipm::ShelfIndex" class="ShelfIndexType"/>

</other-implementations>

</basic-type>

The parameter-type XML file contains :

<parameter-type name="Equipm::Shelf">

<help>the physical position of the shelf</help>

<fields>

<field name="rack" basic-type="Eqpt::Rack" separator="/"/>

<field name="" basic-type="Equipm::ShelfIndex"/>

</fields>

</parameter-type>

The node XML file contains :

<res-id uname="index" parameter-type="Equipm::Shelf">

<help>the physical identification of the shelf</help>

<fields>

<field name="">

<mib-index name="eqptHolderId"/>

</field>

</fields>

</res-id>

Properties :

* The basic-type Equipm::ShelfIndex is another implementation of Eqpt::Shelf which is based on the class ShelfIndexType instead of based on the class SignedIntegerType.
* The basic-type Equipm::ShelfIndex does not have any restrictions like Eqpt::Shelf does. This is not possible because the index calculated by the class is not within the range 1 to 2 as is the case for the plain type Eqpt::Shelf. This means that the class ShelfIndexType must implement special match and expand functions to guarantee that the operator can enter anything else than 1 or 2.
* The field that contains the shelf-number is reused to contain the index, so it gets an empty string as name.
* This field is then further used as mib-index.
  + 1. Object values
       1. When to use ?

Use this pattern when a field contains an object-id and you want to compose this object-id from different fields entered by the operator or display the different components of the object-id.

* + - 1. Basic implementation

In our example is the test result composed from the MIB variable asamAtmVcLoopbackVclVpiResult with the **delay** (an integer) and the **llidr** (a fixed-length binary string of 16 bytes) as indices.

The <Domain>\_tables.cpp file contains :

const MibTableDefinition::MibKeyDefinition atmfM4VcTestResultTableKeys[] = {

{"delay", false},

{"llidr", false}

};

MibTableDefinition atmfM4VcTestResultTable(

"atmfM4VcTestResultTable",

&SnmpAgent::localSnmpAgent\_m,

atmfM4VcTestResultTableKeys,

sizeof(atmfM4VcTestResultTableKeys)/sizeof(atmfM4VcTestResultTableKeys[0]));

The basic-type XML file contains :

<basic-type name="Atm::F5LoopbackTestResult" class="ObjectType">

<help>the result of the requested test</help>

<option>NODEasamAtmVcLoopbackVclVpiResult, &amp;atmfM4VcTestResultTable</option>

</basic-type>

<basic-type name=”Atm::RoundTripDelay” class=”Gauge” unit=”msec”>

<help>the round-trip delay of loopback-test</help>

</basic-type>

<basic-type name=”Atm::Llid” class=”BinaryStringType”>

<help>the logical line identifier</help>

<length min="16" max="16"/>

</basic-type>

The parameter XML file contains :

<parameter-type name=”Atm::RoundTripDelay”>

<help>the round-trip delay of loopback test</help>

<fields>

<field name=”” basic-type=”Atm::RoundTripDelay” version=”1”/>

<field name="result" basic-type=”Atm::F5LoopbackTestResult” version=”32”/>

<field name="llidr" basic-type=”Atm::Llid” version=”32”/>

</fields>

</parameter-type>

The node XML file contains :

<?xml version="1.0" encoding="UTF-8" ?>

<nodes xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="file:///cm4/tools/BELL/xml/xmlCommon/node.xsd"

language="CLI">

<node name="f5-loopback" access="AccessRights::alarm\_r">

<help>show f5-loopback status</help>

<man-page file="CLI\_AlarmReporting/manshowF5Loopback.xml"/>

<default-command>

<command handler-type="show" class="CommandDefinition"/>

</default-command>

<commands>

<command handler-type="help" class="CommandDefinition"/>

<command handler-type="tree" class="CommandDefinition"/>

</commands>

<parameters>

<res-id uname="port" parameter-type="Atm::Vcl">

<help>the port under test</help>

<fields>

<field name="">

<mib-index name="ifIndex"/>

</field>

<field name="vpi">

<mib-index name="atmVclVpi"/>

</field>

<field name="vci">

<mib-index name="atmVclVci"/>

</field>

</fields>

</res-id>

<display-ele name="round-trip-delay" parameter-type="Atm::RoundTripDelay">

<help>the f5-loopback round-trip delay</help>

<fields>

<field name=""

<mib-index name=”delay”/>

</field>

<field name="result" access="CommandFieldDefinition::mayBeReadDuringDisplay\_c">

<mib-var tree-node="NODEatmfM4VcTestResult" table-name="atmfM4VcTestTable"/>

</field>

<field name="llidr" access=" CommandFieldDefinition::mayBeUsedDuringDisplay\_c">

<mib-index name=”llidr” />

</field>

</fields>

</display-ele>

</parameters>

</node>

</nodes>

Properties :

* A table definition atmfM4VcTestResultTable that describes the indices being used : delay and llidr.
* A basic type Atm::F5LoopbackTestResult is created with as class ObjectType.
* This basic type has two options : the first is the reference to the MIB variable NODEasamAtmVcLoopbackVclVpiResult, the second is a pointer to the table definition atmfM4VcTestResultTable.
* An auxiliary field is created in the parameter with as basic type Atm::F5LoopbackTestResult and that is linked to the MIB variable that contains the result.
* The field that must get the value of the delay gets as mib-index delay.
* We add an auxiliary field in the parameter with as mib-index llidr. We use an auxiliary field because the operator is not interested in the value of llidr. We must provide a field with index llidr because otherwise the unpacking of the value will fail.

Note : the given example was for a **show** node, but the same constructs can be used for **configure** nodes.

* + - 1. Variant : extract one single number from an object-id

The following solution can also be used in case we only are interested in one or more numbers in the object-id. It only can be used in **show** commands and does not work for more complex types like binary string. The example given here is an other implementation of the previous example.

The basic-type XML file contains :

<basic-type name=”Atm::RoundTripDelay” class=”Gauge” unit=”msec”>

<help>the round-trip delay of loopback-test</help>

<other-implementations>

<basic-type name=”Atm::ExtractRoundTripDelay” class=”ObjectElementType”>

<option>-17</option>

</basic-type>

</other-implementations>

</basic-type>

The parameter XML file contains :

<parameter-type name=”Atm::ExtractRoundTripDelay”>

<help>the round-trip delay of loopback test</help>

<fields>

<field name=”delay” basic-type=”Atm::ExtractRoundTripDelay” version=”1”/>

<field name=”object” basic-type=”Object” version=”32”/>

</fields>

</parameter-type>

The node XML file contains :

<?xml version="1.0" encoding="UTF-8" ?>

<nodes xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="file:///cm4/tools/BELL/xml/xmlCommon/node.xsd"

language="CLI">

<node name="f5-loopback" access="AccessRights::alarm\_r">

<help>show f5-loopback status</help>

<man-page file="CLI\_AlarmReporting/manshowF5Loopback.xml"/>

<default-command>

<command handler-type="show" class="CommandDefinition"/>

</default-command>

<commands>

<command handler-type="help" class="CommandDefinition"/>

<command handler-type="tree" class="CommandDefinition"/>

</commands>

<parameters>

<res-id uname="port" parameter-type="Atm::Vcl">

<help>the port under test</help>

<fields>

<field name="">

<mib-index name="ifIndex"/>

</field>

<field name="vpi">

<mib-index name="atmVclVpi"/>

</field>

<field name="vci">

<mib-index name="atmVclVci"/>

</field>

</fields>

</res-id>

<display-ele name="round-trip-delay" parameter-type="Atm::ExtractRoundTripDelay">

<help>the f5-loopback round-trip delay</help>

<fields>

<field name="delay">

<master name=”round-trip-delay\_object”/>

</field>

<field name=”object”>

<mib-var tree-node="NODEatmfM4VcTestResult" table-name="atmfM4VcTestTable"/>

</field>

</fields>

</display-ele>

</parameters>

</node>

</nodes>

Properties :

* A new basic-type Atm::ExtractRoundTripDelay is created that is another implementation of Atm::RoundTripDelay.
* The class of Atm::ExtractRoundTripDelay is ObjectElementType.
* The option of Atm::ExtractRoundTripDelay is the offset in the object-id of the number to be extracted. 1 is the first number in the object-id, 2 is the second number, -1 is the last number, -2 is the second last number, … In this case we must specify –17 as the llid is 16 numbers long.
* The parameter-type Atm::ExtractRoundTripDelay contains two fields : the first delay is a field visible to the operator for the delay, the second object is an auxiliary parameter field for the object.
* The object field is linked to the MIB variable.
* The value of the name attribute of the master element of the delay field is the full name of the object field.
  1. Node patterns
     1. Port lists
        1. When to use ?

Use this pattern when each bit set in a binary string indicates that a corresponding object with that number belongs to a list. Typically the operator must be able to add or remove objects from the list.

* + - 1. Basic implementation

In the following example contains the binary string of 24 bits (= 3 bytes). Each bit corresponds to a port number.

A header file **CLI\_**<*Domain*>**/export/LanxVlanBitNode.hpp** is created for a new C++ node class LanxVlanBitNode (note : the name of the class is free, but it is recommended to end the name with **Node**) :

#ifndef \_\_LANXVLANBITNODE\_HPP\_

#define \_\_LANXVLANBITNODE\_HPP\_

#ifndef \_\_PORTLISTNODE\_HPP\_

#include "CLI\_Agent/PortListNode.hpp"

#endif

namespace Cli {

class LanxVlanBitNode : public PortListNode {

public :

LanxVlanBitNode(const char \*name, const char \*help,

const ParameterDefinition \*firstKeyPtr,

const ParameterDefinition \*firstParPtr,

const NodeDefinition \*nextNodePtr,

const NodeDefinition \*firstChildNodePtr,

const CommandDefinition \*firstCommandPtr,

const CommandDefinition \*defaultCommandPtr,

const CommandFieldDefinition \*firstAuxiliaryFieldPtr,

const bool isDynamic,

const size\_t offsetManpage,

const size\_t sizeManpage)

: PortListNode(name, help, firstKeyPtr, firstParPtr, nextNodePtr,

firstChildNodePtr, firstCommandPtr, defaultCommandPtr,

firstAuxiliaryFieldPtr, isDynamic,offsetManpage,

sizeManpage) {}

protected :

virtual size\_t getLastBitPosition() const;

} // NAMESPACE Cli

#endif // \_\_LANXVLANBITNODE\_HPP\_

A source file **CLI\_**<*Domain*>**/LanxVlanBitNode.cpp** is created for this class with following content :

#ifndef \_\_LANXVLANBITNODE\_HPP\_

#include "CLI\_VLAN/LanxVlanBitNode.hpp"

#endif

namespace Cli {

size\_t LanxVlanBitNode::getLastBitPosition() const {

return 23;

}

} // NAMESPACE Cli

The basic-type XML contains :

<basic-type name="Shub::Port" class="SignedIntegerType">

<help>the port number</help>

<restrictions>

<range min-included="1" max-included="24" action="accept" type="signed"/>

</restrictions>

</basic-type>

<basic-type name="Shub::PortList" class="BinaryStringType">

<help>a list of ports (port 1 is the MSB of the first byte)</help>

<length min="3" max="3"/>

</basic-type>

The parameter-type XML contains :

<parameter-type name="Shub::Port">

<help>the port identification</help>

<fields>

<field name="" basic-type="Shub::Port"/>

</fields>

</parameter-type>

The node XML contains :

<?xml version="1.0" encoding="UTF-8"?>

<nodes xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="file:///cm4/tools/BELL/xml/xmlCommon/node.xsd"

language="CLI">

<node name="id" access="AccessRights::vlan\_rw" is-dynamic="true">

<help>configure a specific vlan</help>

<man-page file="./CLI\_VLAN/manLanxConfigureVlanid.xml"/>

<default-command>

<command handler-type="configure" class="CommandDefinition"/>

</default-command>

<commands>

<command handler-type="info" class="CommandDefinition"/>

<command handler-type="help" class="CommandDefinition"/>

<command handler-type="tree" class="CommandDefinition"/>

</commands>

<subnode file="CLI\_VLAN/lanxVlanEgressPortList.xml"/>

<parameters>

<res-id uname="vlanid" parameter-type="Vlan::VlanOamIndex">

<help>vlan id</help>

<fields>

<field name="">

<mib-index name="vlan-index"/>

</field>

</fields>

</res-id>

<fields>

<field name="dot1qVlanStaticRowStatus" basic-type="RowStatus"

access="CommandFieldDefinition::rowStatusAccess\_c">

<mib-var tree-node="NODEdot1qVlanStaticRowStatus"

table-name="dot1qVlanStaticTable" lanx="true"/

</field>

</fields>

</node>

</nodes>

A sub-node XML file lanxVlanEgressPortList.xml is created with following content :

<?xml version="1.0" encoding="UTF-8"?>

<nodes xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="file:///cm4/tools/BELL/xml/xmlCommon/node.xsd"

language="CLI">

<node name="egress-port" access="(AccessRights::AccessGroup)(AccessRights::vlan\_rw)"

class="LanxVlanBitNode" is-dynamic="true">

<help>configure egress ports for specific vlan</help>

<man-page file="./CLI\_VLAN/manLanxVlanEgressPortList.xml"/>

<default-command>

<command handler-type="configure" class="CommandDefinition"/>

</default-command>

<commands>

<command handler-type="info" class="CommandDefinition"/>

<command handler-type="help" class="CommandDefinition"/>

<command handler-type="tree" class="CommandDefinition"/>

</commands>

<parameters>

<res-id uname="port" parameter-type="Shub::Port">

<help>egress port</help>

<fields>

<field name="" access="CommandFieldDefinition::isReadableViaNode\_c|

CommandFieldDefinition::mayBeReadDuringDisplay\_c">

</field>

</fields>

</res-id>

</parameters>

<fields>

<field name="port-list" basic-type="Shub::PortList"

access="CommandFieldDefinition::mayBeReadDuringIn\_c|

CommandFieldDefinition::mayBeReadDuringDisplay\_c|

CommandFieldDefinition::mayBeReadDuringCreate\_c|

CommandFieldDefinition::mayBeReadDuringDelete\_c">

<mib-var tree-node="NODEdot1qVlanStaticEgressPorts"

table-name="dot1qVlanStaticTable" lanx="true"/>

</field>

</fields>

</node>

</nodes>

Properties :

* The new class LanxVlanBitNode inherits from PortListNode.
* The new class LanxVlanBitNode defines the virtual function getLastBitPosition(). As this node must handle the bits 0 to 23, we must here 23. 0 is by default the first bit position. 0 corresponds to the most significant bit of the first byte.
* The new class LanxVlanBitNode is used as node class for the sub-node.
* The sub-node is dynamic.
* The first field of the first res-id port of the sub-node must contain the number of the bit as given by the operator. This number falls in the range 1 (0 + 1) to 24 (23 + 1). This is because by default we let the operator work with numbers, not with offsets.
* This field can get its value via the node class : so the access rights must contain isReadableViaNode. It must be read during display.
* The first auxiliary field port-list of the sub-node must be linked to the MIB variable that contains the bit list. The index for the corresponding MIB table dot1qVlanStaticTable is a field in the parent node.
* The length of the basic type for the port list is set to 3 because it must consist of 3 bytes (it must hold 24 bits).
* The auxiliary field port-list must be read during create, delete, in and display.
  + - 1. Variant : number x does not correspond to bit x-1

Redefine in the node class the functions **bitPositionToNumber()** and **numberToBitPosition()**. Take care that *N* **== bitPositionToNumber(numberToBitPosition(***N***))** is true for each value of *N*.

Don’t forget to adapt the restrictions in the basic-type of the number field accordingly.

* + - 1. Variant : length is larger than can be calculated from the last bit position

Redefine in the node class the function **getBitListSize()** to return the number of bytes of the bit-list.

Don’t forget to adapt the length in the basic-type of the bit-field port-list accordingly.

* + - 1. Variant : the first bit to be handled is not 0

Redefine in the node class the function **getFirstBitPosition()** to return the number of the first bit.

Don’t forget to adapt the restrictions in the basic-type of the number field accordingly.

* + - 1. Variant : createAndWait

Add in the sub-node an auxiliary field with **waitRowStatus** access like in section 5.2.3.4.

Redefine in the node class the function **getRowStatusFieldName()** to return the full name of the new auxiliary field.

* + - 1. Variant : the field with the bit-list is not the first auxiliary field

Redefine in the node class the function **getBitListFieldName()** to return the full name of the field that contains the bit-list.

* + - 1. Variant : the field with the number is not the first field of the first res-id

Redefine in the node class the function **getBitNumberFieldName()** to return the full name of the field that contains the number of the group.

* + - 1. Variant : separate fields must be used to clear and set bits in the bit-list

Add the auxiliary fields to clear and set the bits to the sub-node.

Redefine in the node class the functions **getClearFieldName()** and **getSetFieldName()** to return respectively the full name of the field to clear the bits and to set the bits.

* + 1. Value lists
       1. When to use ?

Use this pattern when a binary string contains X groups of N bits and the operator must be able to change its group separately.

* + - 1. Basic implementation

In following example we have 16 groups of 8 bits.

A header file **CLI\_**<*Domain*>**/export/QosListNode.hpp** is created for a new C++ node class QosListNode (note : the name of the class is free, but it is recommended to end the name with **Node**) :

#ifndef \_\_QOSLISTNODE\_HPP\_

#define \_\_QOSLISTNODE\_HPP\_

#ifndef \_\_VALUELISTNODE\_HPP\_

#include "CLI\_Agent/ValueListNode.hpp"

#endif

namespace Cli {

class QosListNode : public ValueListNode {

public :

QosListNode(const char \*name, const char \*help,

const ParameterDefinition \*firstKeyPtr,

const ParameterDefinition \*firstParPtr,

const NodeDefinition \*nextNodePtr,

const NodeDefinition \*firstChildNodePtr,

const CommandDefinition \*firstCommandPtr,

const CommandDefinition \*defaultCommandPtr,

const CommandFieldDefinition \*firstAuxiliaryFieldPtr,

const bool isDynamic,

const size\_t offsetManpage,

const size\_t sizeManpage)

: ValueListNode(name, help, firstKeyPtr, firstParPtr, nextNodePtr,

firstChildNodePtr, firstCommandPtr, defaultCommandPtr,

firstAuxiliaryFieldPtr, isDynamic,offsetManpage,

sizeManpage) {}

protected :

virtual size\_t getBitsPerGroup() const;

virtual size\_t getLastGroup() const;

} // NAMESPACE Cli

#endif // \_\_QOSLISTNODE\_HPP\_

A source file **CLI\_**<*Domain*>**/QosListNode.cpp** is created for this class with following content :

#ifndef \_\_QOSLISTNODE\_HPP\_

#include "CLI\_VLAN/QosListNode.hpp"

#endif

namespace Cli {

size\_t QosListNode::getBitsPerGroup() const {

return 8;

size\_t QosListNode::getLastGroup() const {

return 15;

}

} // NAMESPACE Cli

The basic-type XML file contains :

<basic-type name="Qos::DscpValue" class="SignedIntegerType">

<help>the DSCP value</help>

<restrictions>

<range min-included="0" max-included="255" action="accept" type="signed"/>

</restrictions>

</basic-type>

<basic-type name="Qos::DscpCodePoint" class="SignedIntegerType">

<help>the number of the DSCP codepoint</help>

<restrictions>

<range min-included="1" max-included="16" action="accept" type="signed"/>

</restrictions>

</basic-type>

<basic-type name=”Qos::ContractTable” class=”ObjectType”>

<help>the DSCP contract table</help>

<length min="16" max="16"/>

</basic-type>

The node XML file contains :

<?xml version="1.0" encoding="UTF-8"?>

<nodes xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="file:///cm4/tools/BELL/xml/xmlCommon/node.xsd"

language="CLI">

<node name="dscp-contract" access="AccessRights::qos\_rw" is-dynamic="true">

<help>configure a marker for a dscp contract table</help>

<man-page file="./CLI\_QoS/manDSCPContract.xml"/>

<default-command>

<command handler-type="configure" class="CommandDefinition"/>

</default-command>

<commands>

<command handler-type="info" class="CommandDefinition"/>

<command handler-type="help" class="CommandDefinition"/>

<command handler-type="tree" class="CommandDefinition"/>

</commands>

<parameters>

<res-id uname="name" parameter-type="AsamProfileName">

<help>A unique profile name</help>

<fields>

<field name="">

<mib-index name=”index”/>

</field>

</fields>

</res-id>

</parameters>

<fields>

<field name="qosMarkerProfileRowStatus" basic-type="RowStatus"

access="CommandFieldDefinition::rowStatusAccess\_c">

<mib-var tree-node="NODEqosMarkerProfileRowStatus"

table-name="qosMarkerProfileTable"/>

</field>

</fields>

</node>

</nodes>

The sub-node XML file contains :

<?xml version="1.0" encoding="UTF-8"?>

<nodes xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="file:///cm4/tools/BELL/xml/xmlCommon/node.xsd"

language="CLI">

<node name="codepoint" access="AccessRights::qos\_rw" class="QosListNode">

<help>configure a codepoint for a dscp contract table</help>

<man-page file="./CLI\_QoS/manqosMarkerProfile.xml"/>

<default-command>

<command handler-type="configure" class="CommandDefinition"/>

</default-command>

<commands>

<command handler-type="info" class="CommandDefinition"/>

<command handler-type="help" class="CommandDefinition"/>

<command handler-type="tree" class="CommandDefinition"/>

</commands>

<parameters>

<res-id uname="codepoint" parameter-type="Qos::DscpCodepoint">

<help>the number of the codepoint</help>

<fields>

<field name="" access="CommandFieldDefinition::isReadableViaNode\_c|

CommandFieldDefinition::mayBeReadDuringDisplay\_c"/>

</fields>

</res-id>

<parameter name="dscp-value" parameter-type="Qos::DscpValue">

<help>the dscp value to be assigned to the codepoint</help>

<optional default="0L"/>

<fields>

<field name="" access="CommandFieldDefinition::isReadableViaNode\_c|

CommandFieldDefinition::isWritableViaNode\_c|

CommandFieldDefinition::mayBeWrittenDuringModify\_c|

CommandFieldDefinition::mayBeReadDuringDisplay\_c"/>

</fields>

</parameter>

</parameters>

<fields>

<field name="value-list" basic-type="Qos::ContractTable"

access="CommandFieldDefinition::mayBeReadDuringPrepare\_c |

CommandFieldDefinition::mayBeWrittenDuringModify\_c |

CommandFieldDefinition::mayBeReadDuringDisplay\_c">

<mib-var tree-node="NODEqosMarkerProfileDscpContractTable"

table-name="qosMarkerProfileTable"/>

</field>

</fields>

</node>

</nodes>

Properties :

* The new class QosListNode inherits from ValueListNode.
* The new class QosListNode defines the virtual function getLastGroup(). As this node must handle the groups 0 to 15, we must here 15. 0 is by default the first group. Group 0 corresponds to the most significant bits in the first byte.
* The new class QosListNode defines the virtual function getBitsPerGroup(). As each group is 8 bits long we return 8.
* The new class QosListNode is used as node class for the sub-node.
* The sub-node is static.
* The first field of the first res-id codepoint of the sub-node must contain the number of the group as given by the operator. This number falls in the range 1 (0 + 1) to 16 (15 + 1). This is because by default we let the operator work with numbers, not with offsets.
* This field can get its value via the node class : so the access rights must contain isReadableViaNode. It must be read during display.
* The first field of the first parameter dscp-value of the sub-node must contain the value of the group. As the value consists of 8 bits the range goes from 0 to 255.
* This field can read and write its value via the node class : so the access rights must contain isReadableViaNode and isWritableViaNode. It must be read during display and written during modify.
* The first auxiliary field value-list of the sub-node must be linked to the MIB variable that contains the value list. The index for the corresponding MIB table qosMarkerProfileTable is a field name in the parent node.
* The length of the basic type for the port list is set to 16 because it must hold 16 groups of 8 bits.
* The auxiliary field value-list must be read during prepare and display and written during modify.
  + - 1. Variant : number x does not correspond to group x-1

Redefine in the node class the functions **groupToNumber()** and **numberToGroup()**. Take care that *N* **== groupToNumber(numberToGroup(***N***))** is true for each value of *N*.

* + - 1. Variant : length is larger than can be calculated from the last group number

Redefine in the node class the function **getBitListSize()** to return the number of bytes of the bit-list.

Don’t forget to adapt the length in the basic-type of the bit-field value-list accordingly

* + - 1. Variant : the first group to be handled is not 0

Redefine in the node class the function **getFirstGroup()** to return the number of the first group.

Don’t forget to adapt the restrictions in the basic-type of the codepoint field accordingly.

* + - 1. Variant : different mapping of a group to bits within a byte

The default mapping is as follows. The example is given for a group of 3 bits :

* Group 0 : bit 7, 6 and 5 of the first byte (7 is the most significant bit)
* Group 1 : bit 4, 3 and 2 of the first byte
* bit 1 and 0 of the first byte are not used
* Group 2 : bit 7, 6 and 5 of the second byte
* …

The mapping can be changed by redefining following functions in the node class :

* **groupToByteOffset()** returns the offset of the byte in which the given group can be found
* **groupToBitsToShift()** returns the number of bits we have to shift the value of the given group to the left so that the group is correctly placed in the byte
  + - 1. Variant : the field with the bit-list is not the first auxiliary field

Redefine in the node class the function **getBitListFieldName()** to return the full name of the field that contains the bit-list.

* + - 1. Variant : the field with the number is not the first field of the first res-id

Redefine in the node class the function **getNumberFieldName()** to return the full name of the field that contains the number of the group.

* + - 1. Variant : the field with the value is not the first field of the first parameter

Redefine in the node class the function **getValueFieldName()** to return the full name of the field that contains the value of the group.

* + 1. Name as res-id instead of index
       1. When to use ?

Use this pattern always when the operator wants to identify instances via a name while the system identifies the instances via unique indices.

Preconditions:

* The system must guarantee that the names are unique.
* The system must provide a MIB or database table that maps a name to an index.
  + - 1. Basic implementation

The basic implementation is for a static node.

The <*Domain*>**\_tables.cpp** file contains :

const MibTableDefinition::MibKeyDefinition domainTableKeys[] = {

{"domainIndex", false}

};

MibTableDefinition domainTable("domainTable",

&SnmpAgent::localSnmpAgent\_m,

domainTableKeys,

sizeof(domainTableKeys)/sizeof(domainTableKeys[0]));

const MibTableDefinition::MibKeyDefinition domainMappingTableKeys[] = {

{"domainMappingName", true}

};

MibTableDefinition domainMappingTable("domainMappingTable",

&SnmpAgent::localSnmpAgent\_m,

domainMappingTableKeys,

sizeof(domainMappingTableKeys)/sizeof(domainMappingTableKeys[0]));

The node XML file contains :

<?xml version="1.0" encoding="UTF-8"?>

<nodes xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="file:///cm4/tools/BELL/xml/xmlCommon/node.xsd"

language="CLI">

<node name="domain" access="AccessRights::aaa\_rw">

<help>configure domain</help>

<man-page file="./CLI\_AAA/manAaaDomain.xml"/>

<default-command>

<command handler-type="configure" class="CommandDefinition"/>

</default-command>

<commands>

<command handler-type="info" class="CommandDefinition"/>

<command handler-type="help" class="CommandDefinition"/>

<command handler-type="tree" class="CommandDefinition"/>

</commands>

<parameters>

<res-id uname="name" parameter-type="Aaa::DomainName">

<help>name of the domain</help>

<fields>

<field name="" access=”CommandFieldDefinition::mayBeReadDuringDisplay\_c”>

<mib-var tree-node="NODEdomainName" table-name="domainTable"/>

<mib-index name="domainMappingName"/>

</field>

</fields>

</res-id>

<parameter name="vlan-id" parameter-type="Vlan::VlanIndexOrZero">

<help>a vlan id</help>

<optional default="(unsigned long)0"/>

<fields>

<field name="">

<mib-var tree-node="NODEdomainVlanIndex" table-name="domainTable"/>

</field>

</fields>

</parameter>

</parameters>

<fields>

<field name="index" basic-type="SignedInteger"

access="CommandFieldDefinition::mayBeReadDuringPrepare\_c|

CommandFieldDefinition::mayBeReadDuringModify\_c|

CommandFieldDefinition::mayBeReadDuringCommit\_c|

CommandFieldDefinition::mayBeReadDuringDelete\_c|

CommandFieldDefinition::mayBeReadDuringIn\_c|

CommandFieldDefinition::mayBeReadDuringDisplay\_c">

<mib-var tree-node="NODEdomainMappingIndex" table-name="domainMappingTable"/>

<mib-index name="domainIndex"/>

</field>

</fields>

</node>

</nodes>

Properties :

* The original table is the domainTable.
* Both the res-id and the parameter fields are columns in this MIB table.
* The res-id field may be read during display.
* The field of the res-id that contains the name is also index in the domainMappingTable.
* The mapping table is domainMappingTable.
* An auxiliary field index is defined. This is a column of the mapping table.
* The auxiliary field is also index in the domainTable.
* The auxiliary field may be read in each applicable phase (create is not applicable because it is a static node).
  + - 1. Variant : dynamic node

Additional support of the system is required. Additionally a scalar MIB variable must be provided that gives the next free index.

The node XML file contains :

<?xml version="1.0" encoding="UTF-8"?>

<nodes xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="file:///cm4/tools/BELL/xml/xmlCommon/node.xsd"

language="CLI">

<node name="domain" access="AccessRights::aaa\_rw" is-dynamic="true">

<help>configure domain</help>

<man-page file="./CLI\_AAA/manAaaDomain.xml"/>

<default-command>

<command handler-type="configure" class="CommandDefinition"/>

</default-command>

<commands>

<command handler-type="info" class="CommandDefinition"/>

<command handler-type="help" class="CommandDefinition"/>

<command handler-type="tree" class="CommandDefinition"/>

</commands>

<parameters>

<res-id uname="name" parameter-type="Aaa::DomainName">

<help>name of the domain</help>

<fields>

<field name="" access="CommandFieldDefinition::mayBeWrittenDuringCreate\_c|

CommandFieldDefinition::mayBeReadDuringDisplay\_c">

<mib-var tree-node="NODEdomainName" table-name="domainTable"/>

<mib-index name="domainMappingName"/>

</field>

</fields>

</res-id>

<parameter name="vlan-id" parameter-type="Vlan::VlanIndexOrZero">

<help>a vlan id</help>

<optional default="(unsigned long)0"/>

<fields>

<field name="">

<mib-var tree-node="NODEdomainVlanIndex" table-name="domainTable"/>

</field>

</fields>

</parameter>

</parameters>

<fields>

<field name="status" basic-type="RowStatus"

access="CommandFieldDefinition::rowStatusAccess\_c">

<mib-var tree-node="NODEdomainRowStatus" table-name="domainTable"/>

</field>

<field name="index" basic-type="SignedInteger"

access="CommandFieldDefinition::mayBeReadDuringPrepare\_c|

CommandFieldDefinition::mayBeReadDuringModify\_c|

CommandFieldDefinition::mayBeReadDuringCommit\_c|

CommandFieldDefinition::mayBeReadDuringDelete\_c|

CommandFieldDefinition::mayBeReadDuringIn\_c|

CommandFieldDefinition::mayBeReadDuringDisplay\_c">

<mib-var tree-node="NODEdomainMappingIndex" table-name="domainMappingTable"/>

<mib-index name="domainIndex"/>

</field>

<field name="next-free" basic-type="SignedInteger"

access="CommandFieldDefinition::mayBeReadDuringCreate\_c">

<mib-var tree-node="NODE\_domainGetNextFreeIndex"

table-name="SnmpAgent::localScalarTable\_m"/>

<mib-index name="domainIndex"/>

</field>

</fields>

</node>

</nodes>

Properties :

* The res-id field may now also be written during create.
* A new auxiliary field is added that is linked to the scalar variable that gives the next free index.
* The mib-index of the new auxiliary field is the SAME as the index of the index auxiliary field.
* The new auxiliary field may only be read during create, this is when the index auxiliary field does not exist.
  + 1. Name as parameter instead of index
       1. When to use ?

Use this pattern when one of the parameters of the command refers to an object that an operator wants to identify via a name, while the system normally identifies it with an index.

* + - 1. Recommended implementation

The recommended implementation is by using the data-driven database approach (see [4]) which is an alternative method to create a database table in the system.

This construct is recommended because :

* It uses the database directly which is much faster than doing the translation via SNMP
* Operator can do expansion on the names
* Operator can request the list of existing names
* Names are verified while the operator types them in

This construct may only be used to read information from the database and never to create, modify or delete rows. These actions must allways be done via SNMP because SNMP supports transactions, logging of changes, … and because the SNMP mappers typically do more than just modifying the database : running semantic checks which are typically not implemented in CLI, creating run-time structures, sending messages, creating entries in other tables, …..

The parameter-type XML file contains :

<parameter-type name="Xdsl::LineProfileName">

<help>the line profile name</help>

<fields>

<datadriven-field name="" basic-type="SignedInteger" datadriven-version="1"

name-column-name="Name" value-column-name="Index"

help-column-name="Descr" access-column-name="Access"

proxy-class="Xdsl::LineProfileProxy">

</datadriven-field>

</fields>

</parameter-type>

Properties:

* The field is now a datadriven-field.
* The basic type depends on the type of the value. In this case the value is a signed integer, so **SignedInteger** was chosen.
* The attribute **proxy-class** contains the name of the database proxy
* The attribute **datadriven-version** must always have the value **1**
* The attribute **name-column-name** contains the name of the database column that contains the name. The name is what the operator must enter (for example: the string **eant-a**).
* The attribute **value-column-name** contains the name of the database column that contains the value. This attribute is optional. The default is the value of the name-column-name attribute. The value is what must be passed as value via SNMP (for example: the string **EANTA**).
* The attribute **help-column-name** contains the name of the database column that contains the description/help information. This attribute is optional. The default is the value of the name-column-name attribute. The help string is the value that will be shown to the operator if he enters a question mark (for example: **the string NT ethernet board electrical netw itf**).
* The attribute **access-column-name** contains the name of the database column that contains the access information (**read**, **write** or **both**). This attribute is optional. Default access is **both**. The access determines if the operator can type in this value or not.
  + - 1. Variant : filtered table

In XML can also be specified to filter out some rows of the table via SQL-like constructs. In following example are all rows selected for which column **CliName** is **not-planned** **or** column **BdGrp** has **not** the value **16**:

<parameter-type name="Equipm::BoardType">

<help>the type of the board</help>

<fields>

<datadriven-field name="" basic-type="PrintableString" datadriven-version="1"

name-column-name="CliName" value-column-name="BdStr"

help-column-name="Descr" access-column-name="CliAcc"

proxy-class="ConfigData::BoardTypeProxy">

<proxy-filter type="or">

<proxy-filter type="not">

<proxy-filter type="equal" name-column-name="BdGrp" column-value="16"/>

</proxy-filter>

<proxy-filter type="equal" name-column-name="CliName" column-value="not-planned"/>

</proxy-filter>

</datadriven-field>

</fields>

</parameter-type>

* + - 1. Fallback implementation

Precondition is that the system provides sufficient support :

* The system must guarantee that the names are unique.
* The system must provide a MIB or database table that maps a name to an index.

The <*Domain*>**\_tables.cpp** file contains :

const MibTableDefinition::MibKeyDefinition radAuthServerTableKeys[] = {

{"server-index",false}

};

MibTableDefinition radAuthServerTable("radAuthServerTable",

&SnmpAgent::localSnmpAgent\_m,

radAuthServerTableKeys,

sizeof(radAuthServerTableKeys)/sizeof(radAuthServerTableKeys[0]));

const MibTableDefinition::MibKeyDefinition radAuthServerMappingTableKeys[] = {

{"server-name",true}

};

MibTableDefinition radAuthServerMappingTable("radAuthServerMappingTable",

&SnmpAgent::localSnmpAgent\_m,

radAuthServerMappingTableKeys,

sizeof(radAuthServerMappingTableKeys)/sizeof(radAuthServerMappingTableKeys[0]));

const MibTableDefinition::MibKeyDefinition radServerSetTableKeys[] = {

{"set-index",false}

};

MibTableDefinition radServerSetTable("radServerSetTable",

&SnmpAgent::localSnmpAgent\_m,

radServerSetTableKeys,

sizeof(radServerSetTableKeys)/sizeof(radServerSetTableKeys[0]));

The node XML file contains :

<?xml version="1.0" encoding="UTF-8"?>

<nodes xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="file:///cm4/tools/BELL/xml/xmlCommon/node.xsd"

language="CLI">

<node name="servers" access="AccessRights::aaa\_rw" is-dynamic=”true”>

<help>configure set of RADIUS servers</help>

<man-page file="./CLI\_AAA/manAaaRadServerSet.xml"/>

<default-command>

<command handler-type="configure" class="CommandDefinition"/>

</default-command>

<commands>

<command handler-type="info" class="CommandDefinition"/>

<command handler-type="help" class="CommandDefinition"/>

<command handler-type="tree" class="CommandDefinition"/>

</commands>

<parameters>

<res-id uname="index" parameter-type="Aaa::RadServerSetIndex">

<help>index of RADIUS profile</help>

<fields>

<field name="" access="CommandFieldDefinition::mayBeReadDuringIn\_c">

<mib-index name="set-index"/>

</field>

</fields>

</res-id>

<parameter name="auth-server" parameter-type="Aaa::RadAuthServer">

<help>the radius authentication server</help>

<fields>

<field name="" access="CommandFieldDefinition::mayBeReadDuringDisplay\_c">

<mib-var tree-node="NODEradAuthServerName" table-name="radAuthServerTable"/>

<mib-index name="server-name"/>

</field>

</fields>

</parameter>

<fields>

<field name="" basic-type="RowStatus"

access="CommandFieldDefinition::rowStatusAccess\_c">

<mib-var tree-node="NODEradServerSetRowStatus" table-name="radServerSetTable"/>

</field>

<field name="mapping-index" basic-type="SignedInteger"

access="CommandFieldDefinition::mayBeReadDuringCreate\_c|

CommandFieldDefinition::mayBeReadDuringPrepare\_c|

CommandFieldDefinition::mayBeReadDuringModify\_c|

CommandFieldDefinition::mayBeReadDuringCommit\_c ">

<mib-var tree-node="NODEradAuthServerMappingIndex"

table-name="radAuthServerMappingTable"/>

</field>

<field name="server-index" basic-type="CopySignedIntegerType"

access="CommandFieldDefinition::mayBeCalculatedDuringCreate\_c|

CommandFieldDefinition::mayBeWrittenDuringCreate\_c|

CommandFieldDefinition::mayBeCalculatedDuringPrepare\_c|

CommandFieldDefinition::mayBeWrittenDuringPrepare\_c|

CommandFieldDefinition::mayBeUsedDuringModify\_c|

CommandFieldDefinition::mayBeUsedDuringCommit\_c|

CommandFieldDefinition::mayBeReadDuringDisplay\_c">

<mib-var tree-node="NODEradServerSetAuthServerIndex"

table-name="radServerSetTable"/>

<mib-index name="server-index"/>

<master name="\_mapping-index"/>

</field>

</fields>

</node>

</nodes>

Properties :

* Three tables are involved : the table returning the server-index when the server-name is given (radAuthServerTable), the table returning the server-name when the server-index is given (radAuthServerMappingTable), the table where the server-index must be filled in (radServerSetTable)
* The field in which the operator enters the name of the server is the index in the radAuthServerMappingTable.
* It is linked to the column in the radAuthServerTable that gives the server-name if the server-index is given.
* An auxiliary field mapping-index contains the index looked up in the radAuthServerMappingTable.
* This auxiliary field mapping-index must be read during create, prepare, modify and commit.
* The value read from the radAuthServerMappingTable must be written in the radServerSetTable. This is done via the server-index field.
* The server-index field must have **CopySignedIntegerType** as basic-type.
* The name of the master element of the server-index field must be the full name of the mapping-index field.
* The class corresponding to **CopySignedIntegerType** has a function **calculateValue()** that copies the value of the master field to itself.
* The function calculateValue() must be executed during the prepare and create phases. This is indicated in the access of the server-index field. Once the value is calculated it must be written too to radServerSetAuthServerIndex in the radServerSetTable. The field must stay available during the modify and commit phases too and must be read during display.
* The server-index field is also the index server-index in the radAuthServerTable table.
  + - 1. Variant : 2 parameters with the same mapping table

In following example must both parameters **auth-server** and **backup-server** use the same translation table. The problem here is that a given mib-index may only be linked to one field. In this case would the mib-index **server-name** be linked to both fields which is not possible.

The <*Domain*>**\_tables.cpp** file contains :

const MibTableDefinition::MibKeyDefinition radAuthServerTableKeys[] = {

{"server-index",false}

};

MibTableDefinition radAuthServerTable("radAuthServerTable",

&SnmpAgent::localSnmpAgent\_m,

radAuthServerTableKeys,

sizeof(radAuthServerTableKeys)/sizeof(radAuthServerTableKeys[0]));

const MibTableDefinition::MibKeyDefinition radAuthServerMappingTableKeys[] = {

{"server-name",true}

};

MibTableDefinition radAuthServerMappingTable("radAuthServerMappingTable",

&SnmpAgent::localSnmpAgent\_m,

radAuthServerMappingTableKeys,

sizeof(radAuthServerMappingTableKeys)/sizeof(radAuthServerMappingTableKeys[0]));

const MibTableDefinition::MibKeyDefinition radServerSetTableKeys[] = {

{"set-index",false}

};

const MibTableDefinition::MibKeyDefinition radAuthServerTable2Keys[] = {

{"server-index2",false}

};

MibTableDefinition radAuthServerTable2("radAuthServerTable",

&SnmpAgent::localSnmpAgent\_m,

radAuthServerTable2Keys,

sizeof(radAuthServerTable2Keys)/sizeof(radAuthServerTable2Keys[0]));

const MibTableDefinition::MibKeyDefinition radAuthServerMappingTable2Keys[] = {

{"server-name2",true}

};

MibTableDefinition radAuthServerMappingTable2("radAuthServerMappingTable",

&SnmpAgent::localSnmpAgent\_m,

radAuthServerMappingTable2Keys,

sizeof(radAuthServerMappingTable2Keys)/sizeof(radAuthServerMappingTable2Keys[0]));

The node XML file contains :

<?xml version="1.0" encoding="UTF-8"?>

<nodes xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="file:///cm4/tools/BELL/xml/xmlCommon/node.xsd"

language="CLI">

<node name="servers" access="AccessRights::aaa\_rw" is-dynamic=”true”>

<help>configure set of RADIUS servers</help>

<man-page file="./CLI\_AAA/manAaaRadServerSet.xml"/>

<default-command>

<command handler-type="configure" class="CommandDefinition"/>

</default-command>

<commands>

<command handler-type="info" class="CommandDefinition"/>

<command handler-type="help" class="CommandDefinition"/>

<command handler-type="tree" class="CommandDefinition"/>

</commands>

<parameters>

<res-id uname="index" parameter-type="Aaa::RadServerSetIndex">

<help>index of RADIUS profile</help>

<fields>

<field name="" access="CommandFieldDefinition::mayBeReadDuringIn\_c">

<mib-index name="set-index"/>

</field>

</fields>

</res-id>

<parameter name="auth-server" parameter-type="Aaa::RadAuthServer">

<help>the radius authentication server</help>

<fields>

<field name="" access="CommandFieldDefinition::mayBeReadDuringDisplay\_c">

<mib-var tree-node="NODEradAuthServerName"

table-name="radAuthServerTable"/>

<mib-index name="server-name"/>

</field>

</fields>

</parameter>

<parameter name="backup-server" parameter-type="Aaa::RadAuthServer">

<help>the backup radius authentication server</help>

<fields>

<field name="" access="CommandFieldDefinition::mayBeReadDuringDisplay\_c">

<mib-var tree-node="NODEradAuthServerName"

table-name="radAuthServerTable2"/>

<mib-index name="server-name2"/>

</field>

</fields>

</parameter>

</parameters>

<fields>

<field name="" basic-type="RowStatus"

access="CommandFieldDefinition::rowStatusAccess\_c">

<mib-var tree-node="NODEradServerSetRowStatus" table-name="radServerSetTable"/>

</field>

<field name="mapping-index" basic-type="SignedInteger"

access="CommandFieldDefinition::mayBeReadDuringCreate\_c|

CommandFieldDefinition::mayBeReadDuringPrepare\_c|

CommandFieldDefinition::mayBeReadDuringModify\_c|

CommandFieldDefinition::mayBeReadDuringCommit\_c ">

<mib-var tree-node="NODEradAuthServerMappingIndex"

table-name="radAuthServerMappingTable"/>

</field>

<field name="server-index" basic-type="CopySignedIntegerType"

access="CommandFieldDefinition::mayBeCalculatedDuringCreate\_c|

CommandFieldDefinition::mayBeWrittenDuringCreate\_c|

CommandFieldDefinition::mayBeCalculatedDuringPrepare\_c|

CommandFieldDefinition::mayBeWrittenDuringPrepare\_c|

CommandFieldDefinition::mayBeUsedDuringModify\_c|

CommandFieldDefinition::mayBeUsedDuringCommit\_c|

CommandFieldDefinition::mayBeReadDuringDisplay\_c">

<mib-var tree-node="NODEradServerSetAuthServerIndex"

table-name="radServerSetTable"/>

<mib-index name="server-index"/>

<master name="\_mapping-index"/>

</field>

<field name="mapping-index2" basic-type="SignedInteger"

access="CommandFieldDefinition::mayBeReadDuringCreate\_c|

CommandFieldDefinition::mayBeReadDuringPrepare\_c|

CommandFieldDefinition::mayBeReadDuringModify\_c|

CommandFieldDefinition::mayBeReadDuringCommit\_c ">

<mib-var tree-node="NODEradAuthServerMappingIndex"

table-name="radAuthServerMappingTable2"/>

</field>

<field name="server-index2" basic-type="CopySignedIntegerType"

access="CommandFieldDefinition::mayBeCalculatedDuringCreate\_c|

CommandFieldDefinition::mayBeWrittenDuringCreate\_c|

CommandFieldDefinition::mayBeCalculatedDuringPrepare\_c|

CommandFieldDefinition::mayBeWrittenDuringPrepare\_c|

CommandFieldDefinition::mayBeUsedDuringModify\_c|

CommandFieldDefinition::mayBeUsedDuringCommit\_c|

CommandFieldDefinition::mayBeReadDuringDisplay\_c">

<mib-var tree-node="NODEradServerSetBackupAuthServerIndex"

table-name="radServerSetTable"/>

<mib-index name="server-index2"/>

<master name="\_mapping-index2"/>

</field>

</fields>

</node>

</nodes>

Properties :

* Create for the second parameter 2 new table definitions radAuthServerTable2 and radAuthServerMappingTable2, pointing to the same MIB tables as the original table definitions
* Both tables have indices that have different names as the original tables : server-name2 and server-index2.
* Two new auxiliary fields are used.
  + - 1. Variant : index has an exception value

The basic-types XML file contains :

<basic-type name="Aaa::IsServer" class="MasterCompareType">

<help>1 if the server index is 0</help>

<option>"0"</option>

</basic-type>

The parameter-types XML file contains :

<parameter-type name="Aaa::RadAuthServerOrNone">

<help>specify a radius authentication server</help>

<fields>

<alternative-field name="present" basic-type="Aaa::IsServer" separator=":">

<alternatives>

<alternative identifier="none" help="no radius server for authentication" value="1L"

version="1"/>

<alternative identifier="name" help="use a radius server for authentication" value="0L"

version="2"/>

</alternatives>

</alternative-field>

<field name="name" basic-type="Aaa::RadAuthServerName" version="2"/>

</fields>

</parameter-type>

The node XML file contains :

<?xml version="1.0" encoding="UTF-8"?>

<nodes xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="file:///cm4/tools/BELL/xml/xmlCommon/node.xsd"

language="CLI">

<node name="servers" access="AccessRights::aaa\_rw" is-dynamic="true">

<help>configure set of RADIUS servers</help>

<man-page file="./CLI\_AAA/manAaaRadServerSet.xml"/>

<default-command>

<command handler-type="configure" class="CommandDefinition"/>

</default-command>

<commands>

<command handler-type="info" class="CommandDefinition"/>

<command handler-type="help" class="CommandDefinition"/>

<command handler-type="tree" class="CommandDefinition"/>

</commands>

<parameters>

<res-id uname="index" parameter-type="Aaa::RadServerSetIndex">

<help>index of RADIUS profile</help>

<fields>

<field name="" access="CommandFieldDefinition::mayBeReadDuringIn\_c">

<mib-index name="radServerSetIndex"/>

</field>

</fields>

</res-id>

<parameter name="auth-server" parameter-type="Aaa::RadAuthServerOrNone">

<help>the radius authentication server</help>

<fields>

<field name="present">

<master name="\_server-index"/>

</field>

<field name="name" access="CommandFieldDefinition::mayBeReadDuringDisplay\_c">

<mib-var tree-node="NODEradAuthServerName" table-name="radAuthServerTable"/>

<mib-index name="server-name"/>

</field>

</fields>

</parameter>

</parameters>

<fields>

<field name="status" basic-type="RowStatus"

access="CommandFieldDefinition::rowStatusAccess\_c">

<mib-var tree-node="NODEradServerSetRowStatus" table-name="radServerSetTable"/>

</field>

<field name="mapping-index" basic-type="SignedInteger"

access="CommandFieldDefinition::mayBeReadDuringCreate\_c|

CommandFieldDefinition::mayBeReadDuringPrepare\_c|

CommandFieldDefinition::mayBeReadDuringModify\_c|

CommandFieldDefinition::mayBeReadDuringCommit\_c">

<mib-var tree-node="NODEradAuthServerMappingIndex"

table-name="radAuthServerMappingTable"/>

</field>

<field name="server-index" basic-type="CopySignedIntegerType"

access="CommandFieldDefinition::mayBeCalculatedDuringCreate\_c|

CommandFieldDefinition::mayBeWrittenDuringCreate\_c|

CommandFieldDefinition::mayBeCalculatedDuringPrepare\_c|

CommandFieldDefinition::mayBeWrittenDuringPrepare\_c|

CommandFieldDefinition::mayBeUsedDuringModify\_c|

CommandFieldDefinition::mayBeUsedDuringCommit\_c|

CommandFieldDefinition::mayBeReadDuringDisplay\_c">

<mib-var tree-node="NODEradServerSetAuthServerIndex"

table-name="radServerSetTable"/>

<mib-index name="server-index"/>

<master name="\_mapping-index"/>

</field>

</fields>

</node>

</nodes>

Properties :

* A new basic type Aaa::IsServer is defined, the class of this basic-type is MasterCompareType.
* The option of this basic type is 0. 0 is the exception value for the server-index : it indicates that there is no server.
* A new parameter type Aaa::RadAuthServer is defined. It consists of two fields : the first present is an alternative with values none and name. None must correspond to the value 1 (= true), name must correspond to 0 (=false). Only after name we can enter the name of the server.
* The present field of the parameter must have the full field name of the server-index field as value for the name attribute of the master element.
* The name field of the parameter is linked to the column of the mapping table as in previous examples.

Execution :

* The class CompareMasterType will in the display phase compare the value of the master field with the value of the option (= 0). If the master value is equal to the option, it will assign 1 to its own field (so that none is displayed), otherwise it will assign 0 (so that name is displayed).
* The class CompareMasterType will, when the command is entered by the operator, compare the own value with 1 (which in our case corresponds to none) and if this is the case assign the value of the option 0 to the master field.
  + 1. Fast ranges
       1. When to use ?

Use this pattern for nodes without parameters and a single res-id of which the value consists of a single numeric field (example : <port-number> but not <rack>/<shelf>/<slot>/<port>).

The advantage for the operator is that :

* Ranges in the command are not expanded to N commands that are executed individually but the range is interpreted by the command itself, which is much faster.
* The output of the info command contains one single line with a range that contains the numbers of all existing subnodes instead of 1 line per existing subnode.
  + - 1. Basic construction

The node XML contains:

<?xml version="1.0" encoding="UTF-8"?>

<nodes xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="file:///cm4/tools/BELL/xml/xmlCommon/node.xsd"

language="CLI">

<node name="fullview-packages"

access="(AccessRights::AccessGroup)(AccessRights::vlan\_rw)"

class="PackageBitNode" is-dynamic="true">

<help>configure package members for specific source</help>

<man-page file="./CLI\_IGMP/manigmpChannelpermission.xml"/>

<default-command>

<command handler-type="configure" class="CommandDefinition"/>

</default-command>

<commands>

<command handler-type="info" class="CommandDefinition"/>

<command handler-type="help" class="CommandDefinition"/>

<command handler-type="tree" class="CommandDefinition"/>

</commands>

<parameters>

<res-id uname="package" parameter-type="Igmp::Pkg">

<help>package member</help>

<fields>

<field name="" access="CommandFieldDefinition::isReadableViaNode\_c|

CommandFieldDefinition::mayBeReadDuringDisplay\_c">

</field>

</fields>

</res-id>

</parameters>

<fields>

<field name="package-list" basic-type="Igmp::PkgList"

access="CommandFieldDefinition::mayBeReadDuringIn\_c|

CommandFieldDefinition::mayBeReadDuringDisplay\_c|

CommandFieldDefinition::mayBeReadDuringCreate\_c|

CommandFieldDefinition::mayBeReadDuringDelete\_c">

<mib-var tree-node="NODEigmpChannelPermPkgBitmap"

table-name="igmpChannelTable"/>

</field>

</fields>

</node>

</nodes>

Properties :

* The construction is an improvement of the port list pattern (see section 6.2.1)
* A node class capable of handing ranges must be linked to the node. The class (in this case **PackageBitNode**) must be constructed in following way:
  + the virtual function **getNextInstance()** of the class may only return 1 instance. A range value representing all existing instances, must then be assigned to the single **res-id** field.
  + the virtual function **compleAction()** of the class must be able to interprete a range value in the **res-id** field.
  + the virtual function **elementHandlesRange()** of the node related class must return **true** to indicate that it is capable of handling the range at the indicated position.
* A type class capable of handling ranges must be linked to the single res-id field (in this case **package\_**). The class (in this case **SignedIntegerRangeType**) must be constructed in following way:
  + the virtual function **convertCommandToCommonRepresentation()** must pass the entered value unchanged to the common value in case it contains a range.
  + the virtual functions **getCommandValue()** and **getShortCommandValue()** must append the common value unchanged to the command value.
  + the virtual function **handlesRange()** of the class must return **true** to indicate that it is capable of handling the range at the indicated position.

1. Patterns for expert cli designers
   1. Validation of strings typed by the operator

This validation is done by the functions **matchToken()** and **expandToken()** found in the type related C++ classes. These functions typically appear together : it is very unlikely that **expandToken()** is not impacted when **matchToken()** is changed.

The **matchToken()** function validates the string entered by the operator.

The **expandToken()** function expands the string entered by the operator as far as possible. It is guaranteed that this function will not be called when **matchToken()** returns **Matcher::noMatch\_c**.

Both functions must return one of following values :

* **Matcher::noMatch\_c** if the entered string is not acceptable.
* **Matcher::partialMatch\_c** if the operator must still enter additional characters to have an acceptable string. This is the value to be returned if the entered string is the empty string.
* **Matcher::expandableMatch\_c** if the entered string is already acceptable, but when the operator may still enter characters to have an acceptable result.
* **Matcher::fullMatch\_c** if the entered string is acceptable but when the entered string will no longer be acceptable when the operator enters any additional character.

The **matchToken()** function must return in **matchedLength** the length of the entered string in case the result is **Matcher::partialMatch\_c**, **Matcher::expandableMatch\_c** or **Matcher::fullMatch\_c**. In case it returns **Matcher::noMatch\_c**, the function must return in **matchedLength** the length of the initial part of the entered string which would result in at least **Matcher::partialMatch\_c**.

1. Index handling
   1. Introduction

MIB tables often contain instances of different kinds.

A typical example is the **ifTable** that contains information on different kinds of interfaces:

* a xdsl-line, identified by rack, shelf, lt-slot, port
* a l2-vlan, identified by rack, shelf, nt-slot, vlan-id
* a software-loopback-interface, identified by rack, shelf, nt-slot
* …

To make it possible to address an instance of the **ifTable** in a uniform way, an index is used. This is a number in which the kind of interface is encoded and all identifiers needed to identify an interface of that kind. The used encoding function can be simple or very complex, depending on the case, but the result must be a number that uniquely identifies a particular interface.

Another example is the **issPortCtrl** table. The indices of this table can be divided in three groups:

* a network port, identified by the network-port-id (value 1 till 7)
* a control port with no identifiers as there is maximum one
* a user port, identified by rack, shelf and lt-slot (value 4 till 19)

The index is a value from 1 till 24 calculated in following way:

* for a network port is the index **17 + network-port-id** (values 18 till 24)
* for the control port is the index **17**
* for the user port is **lt-slot value – 3** (values 1 till 16)

This chapter describes how such indices must be handled in CLI. This methodology was only introduced in R2.4.1 so it is very well possible that you find examples where idices are handled in a different way. It is however the intention that all changes made to CLI (with the exception of simple corrections) are done according to the new methodology described hereafter.

* 1. Nomenclature

In this section, I will define the terms identifier, index-type and index-group.

An index-type identifies an instance of a particular kind. Each index-type has a set of identifiers and a specific encoding/decoding function. In our example of the **issPortCtrl** table are network-port, user-port and control-port index-types.

The numbers used to identify a given index-type, are called identifiers. An identifier is typically a positive number. In our example of the **issPortCtrl** table, are rack, shelf, lt-slot, network-port-id and control-port-id identifiers.

It is obvious that a given identifier can occur in many index-types.

The group of all index-types that can appear together as index of a set of MIB tables is called an index-group. In our example belong the network-port, user-port and control-port to the lanx-port index-group.

An index-type can belong to many index groups, on condition that both the set of identifiers and the result of the encoding are identical.

Sometimes is the set of index-types allowed as index for a given MIB a subset of the index-types allowed for another MIB. In CLI we will use the same index-group for both MIBs.

* 1. Implementation

This design method should only be used for complex indices. Don’t use it if the index-group has only one index-type that has only one identifier. In this case you can use the value of the identifier directly as index.

* + 1. C++ classes
       1. Identifier

In CLI we will provide for each identifier:

* an enumeration value in type **Cli::IndexTranslator::IdentifierType** (header file **CLI\_IndexTranslator/export/IndexTranslator.hpp**)
* a special class that inherits from the abstract class **IndexTranslator** (header file **CLI\_IndexTranslator/export/IndexTranslator.hpp**)

Example: **CLI\_IndexTranslator/export/IndexTranslator.hpp** contains for the lt-slot identifier:

**typedef enum IdentifierType {**

**…**

**ltSlotIdentifier\_c,**

**…**

**} IdentifierType;**

Example: **CLI\_IndexTranslator/IndexTranslator.cpp** contains for the lt-slot identifier:

**#ifndef \_\_LTSLOTINDEXTRANSLATOR\_HPP\_**

**#include "CLI\_IndexTranslator/LtSlotIndexTranslator.hpp"**

**#endif**

…

**IndexTranslator \* IndexTranslator::createIndexTranslator(**

**ResourceManager \*resourceManagerPtr,**

**IdentifierType identifierType) {**

**IndexTranslator \*indexTranslatorPtr = 0;**

**switch (identifierType) {**

…

**case ltSlotIdentifier\_c:**

**indexTranslatorPtr**

**= new(\*resourceManagerPtr) LtSlotIndexTranslator(resourceManagerPtr);**

**break;**

…

**}**

**return indexTranslatorPtr;**

**}**

The identifier specific class defines, apart from the constructor, at least following virtual functions:

* **IdentifierType getIdentifierType() const**: returns the enumeration value of this identifier type
* **bool calculateIdentifierRanges(IdentifierValue &minValue, IdentifierValue &maxValue)**:returns **true** in case the the value of this identifier is relevant (= must be entered by the operator).The values assigned to **minValue** and **maxValue** are respectively the minimum and maximum value allowed for this identifier within the current container: for an lt-slot this is the range of values allowed for lt-slot in the planned shelf.
* **bool calculateExtremeIdentifierRanges(IdentifierValue &minValue, IdentifierValue &maxValue)**: returns **true** in case the the value of this identifier is relevant (= must be entered by the operator).The values assigned to **minValue** and **maxValue** are respectively the minimum and maximum value allowed for this identifier in any possible container. This function will be called in case the container is actually not planned.
* **bool verifyIfPlanned()**: returns **true** in case the current element is planned. This function will only be called in case the container is also planned.

Example: **CLI\_IndexTranslator/LtSlotIndexTranslator.cpp** contains:

**bool LtSlotIndexTranslator::calculateIdentifierRanges(**

**IdentifierValue &minValue,**

**IdentifierValue &maxValue) {**

**bool isRelevant = true;**

**IdentifierValue rack;**

**IdentifierValue shelf;**

**getIdentifier(rackIdentifier\_c, rack);**

**getIdentifier(shelfIdentifier\_c, shelf);**

**if (resourceManagerPtr\_m->mayContinue()) {**

// call some API to retrieve - by preference from the capability matrix - the minimum and

// maximum value that an lt-slot identifier can have in the current shelf and assign the

// values to minValue and maxValue

**minValue = …. ;**

**maxValue = ….;**

**}**

**return isRelevant;**

**}**

**bool LtSlotIndexTranslator::calculateExtremeIdentifierRanges(**

**IdentifierValue &minValue,**

**IdentifierValue &maxValue) {**

**bool isRelevant = true;**

**minValue = 1;**

**maxValue = CLI\_EQUIPMENT\_MAXSLOT - 3;**

**return isRelevant;**

**}**

**bool LtSlotIndexTranslator::verifyIfPlanned() {**

**bool isPlanned = false;**

**IdentifierValue rack;**

**IdentifierValue shelf;**

**IdentifierValue slot;**

**getIdentifier(rackIdentifier\_c, rack);**

**getIdentifier(shelfIdentifier\_c, shelf);**

**getIdentifier(ltSlotIdentifier\_c, slot);**

**if (resourceManagerPtr\_m->mayContinue()) {**

// call some API to verify if the current slot is planned

**isPlanned = …;**

**}**

**return isPlanned;**

**}**

A range of identifiers typically starts at 1. For historical reasons has the first lt-slot number 4 in CLI. To handle this in a proper way we use two numbering schemes: an internal one for CLI starting at 1 and one for the operator starting at 4. All identifier and index-type related procedures work with the internal numbering scheme. A pair of virtual functions need to be defined if the the internal numbering scheme is different from the operator numbering scheme:

* **void userValueToValue(IdentifierValue &value)**: converts a value from operator representation to internal representation
* **void valueToUserValue(IdentifierValue &value)**: converts a value from internal representation to operator representation

Example: **CLI\_IndexTranslator/LtSlotIndexTranslator.cpp** contains:

**void LtSlotIndexTranslator::userValueToValue(IdentifierValue &value) {**

**value -= 3;**

**}**

**void LtSlotIndexTranslator::valueToUserValue(IdentifierValue &value) {**

**value += 3;**

**}**

These functions are defined to make it possible to influence the operator numbering scheme via configuration.

Following additional functions can be defined:

* **IdentifierValue getDefaultValue()**. This function returns the default value for this identifier in case it is not relevant: see **calculateIdentifierRanges()** and **calculateExtremeIdentifierRanges()**. The default implementation returns 1.
* **void valueOutOfRange(IdentifierValue value, IdentifierValue minValue, IdentifierValue maxValue)**. This function displays an error message in case the value entered by the operator is not acceptable.
  + - 1. Index type

In CLI we will provide for each index-type:

* an enumeration value in type **Cli::IndexTypeTranslator::IndexType** (header file **CLI\_IndexTranslator/export/IndexTypeTranslator.hpp**)
* a special class that inherits from the abstract class **IndexTypeTranslator** (header file **CLI\_IndexTranslator/export/IndexTypeTranslator.hpp**)

Example: **CLI\_IndexTranslator/export/IndexTypeTranslator.hpp** contains for the user-port index-type:

**typedef enum IndexType {**

**…**

**userPortIndex\_c,**

**…**

**} IndexType;**

Example: **CLI\_IndexTranslator/IndexTypeTranslator.cpp** contains for the user-port index-type:

**#ifndef \_\_USERPORTINDEXTYPETRANSLATOR\_HPP\_**

**#include "CLI\_IndexTranslator/UserPortIndexTypeTranslator.hpp"**

**#endif**

…

**IndexTypeTranslator \* IndexTypeTranslator::createIndexTypeTranslator(**

**ResourceManager \*resourceManagerPtr,**

**IndexType indexType) {**

**IndexTypeTranslator \*indexTypeTranslatorPtr = 0;**

**switch (indexType) {**

…

**case userPortIndex\_c:**

**indexTypeTranslatorPtr**

**= new(\*resourceManagerPtr) UserPortIndexTypeTranslator(resourceManagerPtr);**

**break;**

…

**}**

**return indexTypeTranslatorPtr;**

**}**

The index-type specific class contains, for each related identifier, an attribute of the class associated with the identifier. Each index-type has at least one such identifier.

The index-type specific class defines, apart from the constructor, at least following virtual functions:

* **IndexType getIndexType() const**: returns the enumeration value of this index type
* **void setIndex(IndexTranslator::IndexValue index)**: analyses the index and assigns values to the identifier attributes.
* **IndexTranslator::IndexValue getIndex()**: calculates the index based on the values of the identifier attributes.

Example: **CLI\_IndexTranslator/export/UserPortIndexTypeTranslator.hpp** contains:

**#ifndef \_\_INDEXTYPETRANSLATOR\_HPP\_**

**#include "CLI\_IndexTranslator/IndexTypeTranslator.hpp"**

**#endif**

**#ifndef \_\_RACKINDEXTRANSLATOR\_HPP\_**

**#include "CLI\_IndexTranslator/RackIndexTranslator.hpp"**

**#endif**

**#ifndef \_\_SHELFINDEXTRANSLATOR\_HPP\_**

**#include "CLI\_IndexTranslator/ShelfIndexTranslator.hpp"**

**#endif**

**#ifndef \_\_LtSlotINDEXTRANSLATOR\_HPP\_**

**#include "CLI\_IndexTranslator/LtSlotIndexTranslator.hpp"**

**#endif**

**class UserPortIndexTypeTranslator : public IndexTypeTranslator {**

**public :**

**UserPortIndexTypeTranslator(ResourceManager \*resourceManagerPtr);**

**virtual IndexTranslator::IndexValue getIndex();**

**virtual void setIndex(IndexTranslator::IndexValue index);**

**virtual IndexTypeTranslator::IndexType getIndexType() const {**

**return IndexTypeTranslator::userPortIndex\_c;**

**}**

**protected :**

**private :**

**RackIndexTranslator rack\_m;**

**ShelfIndexTranslator shelf\_m;**

**LtSlotIndexTranslator slot\_m;**

**};**

Example: **CLI\_IndexTranslator/UserPortIndexTypeTranslator.cpp** contains:

**UserPortIndexTypeTranslator::UserPortIndexTypeTranslator(**

**ResourceManager \*resourceManagerPtr)**

**: IndexTypeTranslator(resourceManagerPtr, &slot\_m) {**

**rack\_m.init(resourceManagerPtr, 0);**

**shelf\_m.init(resourceManagerPtr, &rack\_m);**

**slot\_m.init(resourceManagerPtr, &shelf\_m);**

**}**

**IndexTranslator::IndexValue UserPortIndexTypeTranslator::getIndex() {**

**IndexTranslator::IndexValue index;**

**IndexTranslator::IdentifierValue rack = 0;**

**IndexTranslator::IdentifierValue shelf = 0;**

**IndexTranslator::IdentifierValue slot = 0;**

**getIdentifier(IndexTranslator::shelfIdentifier\_c, shelf);**

**getIdentifier(IndexTranslator::rackIdentifier\_c, rack);**

**getIdentifier(IndexTranslator::ltSlotIdentifier\_c, slot);**

**index = slot;**

**return index;**

**}**

**void UserPortIndexTypeTranslator::setIndex(IndexTranslator::IndexValue index) {**

**if (index <= 16) {**

**setIdentifier(IndexTranslator::rackIdentifier\_c, 1);**

**setIdentifier(IndexTranslator::shelfIdentifier\_c, 1);**

**setIdentifier(IndexTranslator::ltSlotIdentifier\_c, index);**

**} else {**

**identifierIsInvalid(IndexTranslator::rackIdentifier\_c);**

**identifierIsInvalid(IndexTranslator::shelfIdentifier\_c);**

**identifierIsInvalid(IndexTranslator::ltSlotIdentifier\_c);**

**}**

**}**

Note how the identifier attributes are linked to each other in the constructor. The **shelf\_m** attribute gets a pointer to its container: the **rack\_m** attribute. The **slot\_m** attribute gets a pointer to its container: the **shelf\_m** attribute. This linking is crucial to let the index translation module work properly.

It is recommended to implement the function **void getGetNextIndex(IndexTranslator::IndexValue &index)**. This function is especially important for large tables. It avoids that the CLI\_Agent reads a lot of table entries that he will skip anyway because they do not belong to the requested index-type.

This function takes as input an index (not necessary of this index-type) and calculates a new index in such a way that the result of a SNMP get-next with the new value as index is the next row for this index-type. Typically the calculated value is the value of the expected index of this type minus 1.

Example for the user-port index-type:

**void UserPortIndexTypeTranslator::getGetNextIndex(IndexTranslator::IndexValue &index) {**

**if (index >= 16) {**

// after port 16 there are no user ports – jump to the end of the table

**index = 24;**

**}**

**}**

Example for control-port index-type:

**void ControlPortIndexTypeTranslator::getGetNextIndex(IndexTranslator::IndexValue &index) {**

**if (index < 17) {**

// skip the user ports, 17 is the control port

**index = 16;**

**} else {**

// after port 17 there is no other control port – jump to the end of the table

**index = 24;**

**}**

**}**

Example for the network-port index-type:

**void NetworkPortIndexTypeTranslator::getGetNextIndex(IndexTranslator::IndexValue &index) {**

**if (index <= 17) {**

// skip the user and control ports, 18 is the first network port

**index = 17;**

**}**

}

Additionally some functions can be defined that make some properties of identifiers dependant on the index-type. Such dependencies are seldom needed. Implementing these functions should only be done in case it is absolutely necessary:

* **bool calculateIdentifierRanges(IndexTranslator \*indexTranslatorPtr, IndexTranslator::IdentifierValue &minValue, IndexTranslator::IdentifierValue &maxValue);**
* **bool calculateExtremeIdentifierRanges(IndexTranslator \*indexTranslatorPtr, IndexTranslator::IdentifierValue &minValue, IndexTranslator::IdentifierValue &maxValue);**
* **IndexTranslator::IdentifierValue calculateDefaultValue(IndexTranslator \*indexTranslatorPtr);**
* **void userValueToValue(IndexTranslator \*indexTranslatorPtr, IndexTranslator::IdentifierValue &value);**
* **void valueToUserValue(IndexTranslator \*indexTranslatorPtr, IndexTranslator::IdentifierValue &value);**
* **bool isPlanned(IndexTranslator \*indexTranslatorPtr);**
* **void valueOutOfRange(IndexTranslator \*indexTranslatorPtr, IndexTranslator::IdentifierValue value, IndexTranslator::IdentifierValue minValue, IndexTranslator::IdentifierValue maxValue)**

In our case we need to redefine the **isPlanned()** function because a user-port always exists, even if the corresponding lt-slot is not planned.

The file CLI\_IndexTranslator/UserPortIndexTypeTranslator.cpp contains thus following code:

**bool UserPortIndexTypeTranslator::isPlanned(IndexTranslator \*indexTranslatorPtr) {**

**bool isPlanned = false;**

**if (indexTranslatorPtr->getIdentifierType() == IndexTranslator::ltSlotIdentifier\_c) {**

**isPlanned = true;**

**} else {**

// default implementation

**isPlanned = IndexTypeTranslator::isPlanned(indexTranslatorPtr);**

**}**

**return isPlanned;**

**}**

* + - 1. Index group

In CLI we will provide for each index-group:

* an enumeration value in type **Cli::IndexGroup::IndexGroupType** (header file **CLI\_IndexTranslator/export/IndexGroup.hpp**)
* a special class that inherits from the abstract class **IndexGroup** (header file **CLI\_IndexTranslator/export/IndexGroup.hpp**)
* a static instance of the index-group class (header file **CLI\_IndexTranslator/export/IndexGroup.hpp**)

Example: **CLI\_IndexTranslator/export/IndexGroup.hpp** contains for the lanx-port index-group:

**class LanxPortIndexGroup;**

…

**typedef enum IndexGroupType {**

**…**

**lanxPortIndexGroup\_c,**

**…**

**} IndexGroupType;**

**…**

**static LanxPortIndexGroup lanxPortIndexGroup\_m;**

Example: **CLI\_IndexTranslator/IndexGroup.cpp** contains for the lanx-port index-group:

**#ifndef \_\_LANXPORTINDEXGROUP\_HPP\_**

**#include "CLI\_IndexTranslator/LanxPortIndexGroup.hpp"**

**#endif**

**…**

**LanxPortIndexGroup IndexGroup::lanxPortIndexGroup\_m;**

**…**

**IndexGroup \*IndexGroup::getIndexGroup(ResourceManager &resourceManager,**

**IndexGroupType indexGroupType) {**

**…**

**case lanxPortIndexGroup\_c:**

**indexGroupPtr = &lanxPortIndexGroup\_m;**

**break;**

**…**

**}**

The index-group specific class defines, apart from the constructor, one virtual function **getIndexType()**. This function gives the index-type for a given index.

Example: **CLI\_IndexTranslator/LanxPortIndexGroup.cpp** contains:

**IndexTypeTranslator::IndexType LanxPortIndexGroup::getIndexType(**

**IndexTranslator::IndexValue index,**

**ResourceManager &resourceManager) const {**

**IndexTypeTranslator::IndexType indexType;**

**indexType = IndexTypeTranslator::noIndex\_c;**

**if (index < 17) {**

**indexType = IndexTypeTranslator::userPortIndex\_c;**

**} else if (index > 17) {**

**indexType = IndexTypeTranslator::networkPortIndex\_c;**

**} else {**

**indexType = IndexTypeTranslator::controlPortIndex\_c;**

**}**

**return indexType;**

}

* + 1. XML files
       1. The index type is explicit – all index-types are allowed

In this case is the index-type visible (=explicit) in the parameter value. Example: **lt:1/1/4**

Create a basic-type for each identifier. The **class** attribute must be **IdentifierType**, the **option** tag must contain the enumeration value corresponding to this identifier. Note that no restrictions must be defined, as they are calculated at run-time:

**<basic-type name="Eqpt::RackId" class="IdentifierType">**

**<help>the rack number</help>**

**<option>IndexTranslator::rackIdentifier\_c</option>**

**</basic-type>**

**<basic-type name="Eqpt::ShelfId" class="IdentifierType">**

**<help>the shelf number</help>**

**<option>IndexTranslator::shelfIdentifier\_c</option>**

**</basic-type>**

**<basic-type name="Eqpt::SlotId" class="IdentifierType">**

**<help>the slot number</help>**

**<option>IndexTranslator::ltSlotIdentifier\_c</option>**

**</basic-type>**

**<basic-type name="Shub::NetworkPort" class="IdentifierType">**

**<help>network port on Shub</help>**

**<option>IndexTranslator::networkPortIdentifier\_c</option>**

**</basic-type>**

**<basic-type name="Shub::ControlPort" class="IdentifierType">**

**<help>control port on Shub</help>**

**<option>IndexTranslator::controlPortIdentifier\_c</option>**

**</basic-type>**

**<basic-type name="Shub::UserPort" class="IdentifierType">**

**<help>control port on Shub</help>**

**<option>IndexTranslator::controlPortIdentifier\_c</option>**

**</basic-type>**

Create a basic type for the index-group. The **class** attribute used here is **SignedIndexType**. This class can be used for all indices that are defined in the MIB as signed integer. Use **UnsignedIndexType** for indices that are defined in the MIB as unsigned integers. The **option** tag must contain the enumeration value corresponding to the index-group. Note that no restrictions must be defined.

**<basic-type name="Shub::PortIndex" class="SignedIndexType">**

**<help>an index of the lanx port group</help>**

**<option>IndexGroup::lanxPortIndexGroup\_c</option>**

**</basic-type>**

The parameter type files for our example contain:

**<parameter-type name="Shub::Port">**

**<help>the port identification</help>**

**<fields>**

**<alternative-field name="type" basic-type="IndexType" separator=":">**

**<alternatives>**

**<alternative identifier="lt" help="port of the line board"**

**value="(long)IndexTypeTranslator::userPortIndex\_c" version="1"/>**

**<alternative identifier="network" help="network port"**

**value="(long)IndexTypeTranslator::networkPortIndex\_c" version="2"/>**

**<alternative identifier="nt" help="port of the nt"**

**value="(long)IndexTypeTranslator::controlPortIndex\_c" version="3"/>**

**</alternatives>**

**</alternative-field>**

**<field name="rack" basic-type="Eqpt::RackId" separator="/" version="1"/>**

**<field name="shelf" basic-type="Eqpt::ShelfId" separator="/" version="1"/>**

**<field name="slot" basic-type="Eqpt::SlotId" separator="/" version="1"/>**

**<field name="network" basic-type="Shub::NetworkPort" version="2"/>**

**<field name="control" basic-type="Shub::ControlPort" version="32"/>**

**<field name="" basic-type="Shub::PortIndex" version="32"/>**

**</fields>**

**</parameter-type>**

The index-type is mentioned explicitely via the alternative field. The basic type of the index is the one corresponding to the index-group. The index itself is the last field and is always defined as an auxiliary field (version 32). The identifier fields come after the alternative field. Note that in **Shub::Port** also a field is defined for the control-port identifier but that it is an auxiliary field (version 32). This indicates that an operator never has to give it a value (the CLI\_Agent will assign a default value).

* + - 1. The index type is explicit – only some index-types are allowed

In following example are only network-port index-types allowed. The CLI will automatically filter out all other possible index-types based on the possible values of the alternative-field.

**<parameter-type name="Shub::FullNetworkPort">**

**<help>a network port</help>**

**<fields>**

**<alternative-field name="type" basic-type="IndexType" separator=":">**

**<alternatives>**

**<alternative identifier="network" help="network port"**

**value="(long)IndexTypeTranslator::networkPortIndex\_c" version="1"/>**

**</alternatives>**

**</alternative-field>**

**<field name="network" basic-type="Shub::NetworkPort" version="1"/>**

**<field name="" basic-type="Shub::PortIndex" version="32"/>**

**</fields>**

**</parameter-type>**

* + - 1. The index type is implicit

In this case is the index type not visible (= implicit) in the parameter value. The index type is known from the context. Example : **configure interface shub port 7**.

In this case only index-type may be allowed. The allowed index-type is mentioned explicitely as second option for the basic type of the index-group:

**<basic-type name="Shub::NetworkPortIndex" class="SignedIndexType">**

**<help>an index of the lanx port group</help>**

**<option>IndexGroup::lanxPortIndexGroup\_c,**

**IndexTypeTranslator::networkPortIndex\_c</option>**

**</basic-type>**

The parameter type becomes then:

**<parameter-type name="Shub::NetworkPort">**

**<help>isam network port</help>**

**<fields>**

**<field name="network" basic-type="Shub::NetworkPort" version="2"/>**

**<field name="" basic-type="Shub::NetworkPortIndex" version="32"/>**

**</fields>**

**</parameter-type>**

1. backwards compatibility
   1. Applicability

Our customers require that CLI commands used in scripts are backwards compatible: these are all commands in the **configure** node. Backward compatibility is not guaranteed for commands in the other nodes (for example: **admin** or **show**).

Also the output of commands (**info**, **tree**, **show**) may change between releases. Tools should always use the output in XML format because this is the only format for which backwards compatibility is guaranteed.

* 1. Avoiding backward compatibility problems

The best way to avoid problems is to think ahead.

Try to forcast how the command will evolve in following releases:

* Don’t use a boolean parameter if the attribute has now two possible values but when it could have more values in future (for example: now **active** and **inactive**, in a future release also **hot-standby**).
* Provide a res-id in a node even if you can have at this moment only one instance when you expect there could be multiple instances in the future (for example: management vlan)
* Use an explicit type-identifier if it is possible that in future the value of a parameter could refer to other kinds of objects (for example: **port** **lt:1/1/4** instead of **port 1/1/4** if in future **port** **network:2** would also be possible)
  1. Solutions in case of incompatibilities
     1. Obsolete parameters
        1. Unused parameters

It can occur that a parameter becomes obsolete because the corresponding function has been removed from the system. Example: it could be that in future releases we only support ECNT-C-like boards where the command to reboot the SHUB alone does not make any sense because SHUB and NT share the same processor.

The following must be done for such parameter:

* An **obsolete** tag is added to the parameter definition.
* All **fields** and **optional** tags must be removed in case om a “**named**” parameter
* The optional tags may not be removed in case of an “**unnamed**” parameter! Only remove the **fields**.

Old XML file (example *named* parameter):

<parameter name="reboot-shub" parameter-type="Sys::RestartValue">

<help>reboot the shub</help>

<optional/>

<fields>

<field name="">

<mib-var tree-node="NODE\_issRestart" table-name="SnmpAgent::lanxScalarTable\_m"

lanx="true"/>

</field>

</fields>

</parameter>

New XML file:

<parameter name="reboot-shub" parameter-type="Sys::RestartValue">

<help>reboot the shub</help>

<obsolete/>

</parameter>

Old XML file (example *unnamed* parameter):

<parameter uname="proprietary-feat" parameter-type="Xdsl::FeatureDisable">

<help>enable proprietary features of the modem</help>

<optional default="&quot;no-proprietary&quot;"/>

<fields>

<field name="">

<mib-var tree-node="NODEadsl2PlusModemFeatures" table-name="xdslLineSpectrumProfileAdsl2PlusTable"/>

</field>

</fields>

</parameter>

New XML file:

<parameter uname="proprietary-feat" parameter-type="Xdsl::FeatureDisable">

<help>enable proprietary features of the modem</help>

<optional default="&quot;no-proprietary&quot;"/>

<obsolete/>

</parameter>

Effect for the operator:

* the manual indicates that the parameter is obsolete
* the parameter is marked in the one-line help with an **X** to indicate that it is obsolete
* nothing will happen if the command is entered by the operator
* the parameter will never appear in the output of an info command
  + - 1. Replaced parameters

A typical case is a parameter that was originally implemented as a boolean, but where more different values must be accepted in the current release.

The following must be done for such parameter:

* An **obsolete** tag with the attribute is-replaced-by=”next” must be added to the parameter definition.
* The replacing parameter must be put in XML file directly after the replaced parameter.

Old XML:

<parameter uname="fixed-format" parameter-type="Pppoe::FixedFormat">

<help>set the format of the pppoe relay tag to fixed</help>

<optional default=”&quot;none&quot;”/>

<fields>

<field name="">

<mib-var tree-node="NODEextendAddPppoeRelayTag"

table-name="extendVlanStaticTable"/>

</field>

</fields>

</parameter>

New XML:

<parameter uname="fixed-format" parameter-type="Pppoe::FixedFormat">

<help>set the format of the pppoe relay tag to fixed</help>

<obsolete replaced-by=”next”/>

<optional default=”&quot;none&quot;”/>

<fields>

<field name="">

<mib-var tree-node="NODEextendAddPppoeRelayTag"

table-name="extendVlanStaticTable"/>

</field>

</fields>

</parameter>

<parameter name="relay-tag" parameter-type="Pppoe::RelayTag">

<help>the format of the pppoe relay tag</help>

<optional default=”&quot;none&quot;”/>

<fields>

<field name="">

<mib-var tree-node="NODEextendAddPppoeRelayTag"

table-name="extendVlanStaticTable"/>

</field>

</fields>

</parameter>

In case multiple parameters are replaced with one parameter, then all replaced parameters must be placed after each other in the XML file, directly followed by the replacing parameter. All replaced parameters must have the **obsolete** tag with the attribute is-replaced-by=”next”.

Effect for the operator:

* the manual indicates that the replaced parameters are obsolete
* the replaced parameters are marked in the one-line help with an **X** to indicate that they are obsolete
* a replaced parameter can not be entered anymore once the replacing parameter is entered
* the replacing parameter can not be entered anymore once a replaced parameter is entered
* the replaced parameter will never appear in the output of an info command
  + 1. Obsolete alternatives

Sometimes it can occur that the name of an alternative has to change, for example because the system in the current release interpretes the corresponding value in a different way than the system in the previous releases or because the chosen name is just not good.

In the following example is the alternative with name **user** replaced by the next alternative lt. Note that the **replaced-by** attribute is mandatory.

**<alternative-field name="type" basic-type="IndexType" separator=":">**

**<alternatives>**

**<alternative identifier="user" help="userport"**

**value="(long)IndexTypeTranslator::userPortIndex\_c" version="1">**

**<obsolete replaced-by=”next”/>**

**</alternative>**

**<alternative identifier="lt" help="port of the line board"**

**value="(long)IndexTypeTranslator::userPortIndex\_c" version="1"/>**

**<alternative identifier="network" help="network port"**

**value="(long)IndexTypeTranslator::networkPortIndex\_c" version="2"/>**

**<alternative identifier="nt" help="port of the nt"**

**value="(long)IndexTypeTranslator::controlPortIndex\_c" version="3"/>**

**</alternatives>**

**</alternative-field>**

Effect for the operator:

* the manual indicates that the alternative is obsolete
* the alternative is marked in the one-line help with an **X** to indicate that it is obsolete
* the alternative will never appear in the output of an info command
  + 1. Obsolete commands

Sometimes a complete command becomes obsolete.

Let us take following command as example:

**<?xml version="1.0" encoding="UTF-8"?>**

**<nodes xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"**

**xsi:noNamespaceSchemaLocation="file:///cm4/tools/BELL/xml/xmlCommon/node.xsd"**

**language="CLI">**

**<node name="tcp-filter" access="AccessRights::ip\_rw" is-dynamic="true">**

**<help>configure a filter</help>**

**<man-page file="CLI\_IP/manTcpFilter.xml"/>**

**<default-command>**

**<command handler-type="configure" class="CommandDefinition"/>**

**</default-command>**

**<commands>**

**<command handler-type="info" class="CommandDefinition"/>**

**<command handler-type="help" class="CommandDefinition"/>**

**<command handler-type="tree" class="CommandDefinition"/>**

**</commands>**

**<parameters>**

**<res-id uname="tcp-filter" parameter-type="Ip::TcpFilterId">**

**<help>the number of the tcp-filter</help>**

**<fields>**

**<field name="">**

**<mib-index name="filter-id"/>**

**</field>**

**</fields>**

**</res-id>**

**<parameter name="port" parameter-type="Ip::TcpPortNumber">**

**<help>the port that must be filtered</help>**

**<optional default=”23”/>**

**<fields>**

**<field name="">**

**<mib-var tree-node="NODEfilterPort" table-name="filterTable"/>**

**</field>**

**</fields>**

**</parameter>**

**</parameters>**

**<fields>**

**<field name="status" basic-type="RowStatus"**

**access="CommandFieldDefinition::rowStatusAccess\_c">**

**<mib-var tree-node="NODEfilterRowStatus"**

**table-name="filterTable"/>**

**</field>**

**</fields>**

**</node>**

**</nodes>**

* + - 1. Unused command

In this case the command becomes obsolete because the corresponding functionality is removed from the system.

The following must be done for such command:

* An **obsolete** tag is added to the node definition.
* All **fields** and **optional** tags must be removed.
* The **info** **handler-type** tag is removed from the **commands** tag.
* The **node class** and related **node class header** must be removed.

The XML file then becomes:

**<?xml version="1.0" encoding="UTF-8"?>**

**<nodes xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"**

**xsi:noNamespaceSchemaLocation="file:///cm4/tools/BELL/xml/xmlCommon/node.xsd"**

**language="CLI">**

**<node name="tcp-filter" access="AccessRights::ip\_rw" is-dynamic="true">**

**<help>configure a filter</help>**

**<obsolete/>**

**<man-page file="CLI\_IP/manTcpFilter.xml"/>**

**<default-command>**

**<command handler-type="configure" class="CommandDefinition"/>**

**</default-command>**

**<commands>**

**<command handler-type="help" class="CommandDefinition"/>**

**<command handler-type="tree" class="CommandDefinition"/>**

**</commands>**

**<parameters>**

**<res-id uname="tcp-filter" parameter-type="Ip::TcpFilterId">**

**<help>the number of the tcp-filter</help>**

**</res-id>**

**<parameter name="port" parameter-type="Ip::TcpPortNumber">**

**<help>the port that must be filtered</help>**

**</fields>**

**</parameter>**

**</parameters>**

**</node>**

**</nodes>**

Effect for the operator:

* executing the command will have no effect
* the manual indicates that the command is obsolete
* the command is marked in the one-line help with an **X** to indicate that it is obsolete
* the command will never appear in the output of an info command
  + - 1. Replaced command

In this case the command becomes obsolete because the it is replaced by a new command.

The following must be done for such command:

* An **obsolete** tag is added to the node definition and the replaced-by attribute specifies the absolute path of the replacing command. If the functionality of the command is spread over more than one command, then a list of the absolute paths of all replacing commands must be specified, each path separated from the next with a comma.
* The **info** **handler-type** tag is removed from the **commands** tag.

Let us take following command as example:

**<?xml version="1.0" encoding="UTF-8"?>**

**<nodes xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"**

**xsi:noNamespaceSchemaLocation="file:///cm4/tools/BELL/xml/xmlCommon/node.xsd"**

**language="CLI">**

**<node name="tcp-filter" access="AccessRights::ip\_rw" is-dynamic="true">**

**<help>configure a filter</help>**

**<obsolete replaced-by=”configure ip filter”/>**

**<man-page file="CLI\_IP/manTcpFilter.xml"/>**

**<default-command>**

**<command handler-type="configure" class="CommandDefinition"/>**

**</default-command>**

**<commands>**

**<command handler-type="help" class="CommandDefinition"/>**

**<command handler-type="tree" class="CommandDefinition"/>**

**</commands>**

**<parameters>**

**<res-id uname="tcp-filter" parameter-type="Ip::TcpFilterId">**

**<help>the number of the tcp-filter</help>**

**<fields>**

**<field name="">**

**<mib-index name="filter-id"/>**

**</field>**

**</fields>**

**</res-id>**

**<parameter name="port" parameter-type="Ip::TcpPortNumber">**

**<help>the port that must be filtered</help>**

**<optional default=”23”/>**

**<fields>**

**<field name="">**

**<mib-var tree-node="NODEfilterPort" table-name="filterTable"/>**

**</field>**

**</fields>**

**</parameter>**

**</parameters>**

**<fields>**

**<field name="status" basic-type="RowStatus"**

**access="CommandFieldDefinition::rowStatusAccess\_c">**

**<mib-var tree-node="NODEfilterRowStatus"**

**table-name="filterTable"/>**

**</field>**

**</fields>**

**</node>**

**</nodes>**

Effect for the operator:

* the manual indicates that the command is obsolete
* the manual indicates which command(s) replaces the command
* the command is marked in the one-line help with an **X** to indicate that it is obsolete
* the command will never appear in the output of an info command
  1. Addition of explicit type-identifier
     1. When to use ?

Use this solution in case no type-indication was provided in the old implementation of the parameter and when it must be possible in the new implementation to address different kinds of objects.

* + 1. Basic implementation

In the following example we want to give the operator the possibility to use a TL1 like numbering scheme to address a slot (**lt:**<*rack*>**/**<*shelf*>**/**<*lt-slot*>, **nt-a**, **nt-b** or **acu:**<*rack*>**/**<*shelf*>) beside the traditional CLI numbering scheme <*rack*>**/**<*shelf*>**/**<*eq-slot*>.

The original XML file:

<parameter-type name="Equipm::SlotIndex">

<help>the physical position of the slot</help>

<fields>

<field name="rack" basic-type="Eqpt::RackId" separator="/" version="1"/>

<field name="shelf" basic-type="Eqpt::ShelfId" separator="/" version="1"/>

<field name="eq-slot" basic-type="Eqpt::EqSlotId" version="1"/>

<field name="" basic-type="Eqpt::SlotIndex" version="32"/>

</fields>

</parameter-type>

The modified XML file:

<parameter-type name="Equipm::SlotIndex">

<help>the physical position of the slot</help>

<fields>

<alternative-field name="type" basic-type="IndexType" separator=":">

<alternatives>

<alternative identifier="lt" help="lt-slot"

value="(long)IndexTypeTranslator::ltEquipmentIndex\_c" version="1"/>

<alternative identifier="nt-a" help="nt-a slot"

value="(long)IndexTypeTranslator::ntAEquipmentIndex\_c" version="2"/>

<alternative identifier="nt-b" help="nt-b slot"

value="(long)IndexTypeTranslator::ntBEquipmentIndex\_c" version="2"/>

<alternative identifier="acu" help="acu slot"

value="(long)IndexTypeTranslator::acuEquipmentIndex\_c" version="3"/>

<invisible value="(long)IndexTypeTranslator::slotEquipmentIndex\_c" version="4"/>

</alternatives>

</alternative-field>

<field name="rack" basic-type="Eqpt::RackId" separator="/" version="1,3,4"/>

<field name="shelf" basic-type="Eqpt::ShelfId" separator="/" version="1,3,4"/>

<field name="lt-slot" basic-type="Eqpt::SlotId" version="1"/>

<field name="eq-slot" basic-type="Eqpt::EqSlotId" version="4"/>

<field name="nta-slot" basic-type="Eqpt::NtASlotId" version="32"/>

<field name="ntb-slot" basic-type="Eqpt::NtBSlotId" version="32"/>

<field name="acu-slot" basic-type="Eqpt::AcuSlotId" version="32"/>

<field name="" basic-type="Eqpt::SlotIndex" version="32"/>

</fields>

</parameter-type>

Properties:

* Add an alternative field with TL1-like prefices
* Add an invisible alternative to the alternative field. An invisible alternative matches only when it is followed by a numeric character, but it has itself no length.
* Add the new fields for the TL1-like identifiers.

A version refers to a certain combination of fields that are visible to the operator. As such it is correct that nt-a and nt-b share the same version number 2 as in both cases the type field is the only visible field.

1. Build
   1. Build products

These are the special build products :

* The user manual in PDF format : **CLI\_Language/userguide.pdf**
* The user manual in HTML format : **CLI\_Language/userguide.html**
* The man page file required for the **help** command : **CLI\_Language/manpage.txt**
* A list of all commands with parameters : **CLI\_Language/CLICommands.txt**
* An overview of all used XML files : **CLI\_Language/filetree.txt**
* All node files merged : **CLI\_Language/nodeMerge.xml**
* All parameter type files merged : **CLI\_Language/parameterTypesMerged.xml**
* All basic type files merged : **CLI\_Language/basicTypesMerged.xml**
* All handler files merged : **CLI\_Language/handlerMerge.xml**
* All printer files merged : **CLI\_Language/printerMerge.xml**
* All filter files merged : **CLI\_Language/filterMerge.xml**

Note : the **userguide.pdf**,**userguide.html** and **filetree.txt** files are not produced if the environment variable **NO\_MANUAL** is defined and has another value than 0. This environment variable is often set to reduce the build time.

* 1. MIBCO

1. Host TESTING
   1. Help file

The help command will not work on host unless you create a symbolic link to the **allman.txt**.

You can do this by executing following commands :

* **cd** <*HostTestDir*>/**rack01subr01/slot01/Sw**
* **rm MERYAA00.001**
* **ln –s /view/**<*View*>**/vobs/dsl/source08/CLI\_Language/manpage.txt MERYAA00.001**

*HostTestDir* is the host test directory (typically located in the user directory).

*View* is the name of the view in which the man page file was build.

* 1. Simulated environment

Often we want to test our CLI commands when the corresponding SNMP mapper is not yet implemented.

It is now possible (starting from R2.5 and R3.1 on) to use a simulated context. This is a very simple mapper that stores in a hash-table the values:

* set via an SNMP set-request (via an external manager or via CLI)
* programmed via trace and debug

It will return this values on following SNMP get and get-next requests. It will return **noSuchInstance** when you try to get an entry for which no value was stored. It is also possible to provoke an SNMP error on get or set requests.

The related trace and debug commands are:

* **snmp simulate delete** *object-id* to delete a stored entry
* **snmp simulate set** *object-id value* to store an entry
* **snmp simulate read-error** *object-id* error to provoke an snmp error on a read request
* **snmp simulate write-error** *object-id* error to provoke an snmp error on a write request
  + *object-id* has the form *mib-var*.*indices*, example: **sysContact.0**
  + the format of *value* depends on the type of the mib-variable
    - an unsigned decimal integer for **VB\_gauge**, **VB\_timeTicks**, **VB\_counter**, **VB\_counter64**, **VB\_gauge64**
    - a signed decimal integer for **VB\_integer**
    - an ip-address for **VB\_ipAdr**
    - *mib-var*.*indices* for an **VB\_objId**
    - a quoted string or *hex*:*hex*:*hex*:*hex* for a **VB\_octetStr** or **VB\_opaque**

To sent an SNMP message:

* with SNMPv1: create a SNMP community string linked to context **sim** (**configure system security snmp community**)
* with SNMPv3: use directly the context **sim**

To use the simulated context for the CLI commands that you want to test:

* replace for table objects in the definition of your MIB table (**MibTableDefinition)** the reference to **SnmpAgent::localSnmpAgent\_m** or **SnmpAgent::lanxSnmpAgent\_m** with **SnmpAgent::simulatedSnmpAgent\_m**.
* replace for scalar objects the reference to **SnmpAgent::localScalarTable\_m** or **SnmpAgent::lanxScalarTable\_m** with **SnmpAgent::simulatedScalarTable\_m.**