Microchip

SNMP/JSON Programmers Guide

for release 4.6
January 31th, 2019

1 Private MIB top-down

In order to follow this description, you should have the snmp demo files which is available as a demo_snmp.tar.gz file. It shall be unpacked in the `vtss_appl` directory, i.e.

```
$ cd .../vtss_appl
$ tar xzf demo_snmp.tar.gz
```

Then there should be a directory called `vtss_appl/demo_snmp`.

This code illustrate 5 examples

- Ex1 with a single get/set attribute
- Ex2 with a single get attribute with a trap
- Ex3 a simple table with an integer index
- Ex4 a simple table with an ifIndex index
- Ex5 a table with 3 index attributes
- Ex6 a table that emit traps on add, delete and modify operations
- Ex7 a table that is can be written to from SNMP

For tables, example 3 is the main example. It is not more complicated than it need to be. If this example is understood, then example 4 and 5 are the same thing, with a bit more details, which would be confusing to learn in a first round.

1.1 Introduction to the examples

In the `demo_mib.cxx` the hook-up to SNMP is done[1]. Search for `_demo_ex1`. This object represent the SNMP object that is a simple get/set attribute. The actual implementation of the attribute is done in `DemoEx1`, which the class template used to define `_demo_ex1` is a parameter to. In the next section we will look at how that is supposed to be done. For now we assume it is done, and just focus at how the attribute comes into the SNMP tree.

If we look at the important code for locating our object in the SNMP tree it will look like this

```
VTSS_MIB_MODULE("DemoMib", "DEMO", demo_mib_init, VTSS_MODULE_ID_DEMO,
                demo_root, h) {...}
NS(demo_mib_objects, demo_root, 11, "DemoMibObjects");
NS(demo_status,    demo_mib_objects, 13, "DemoStatus");
static StructRW2<DemoEx1>       _demo_ex1(
     &demo_status, expose::snmp::OidElement(17, "DemoEx1"));
```

then the `OID` of `_demo_ex1` will be

```
enterprises.vtss.vtssSwitchMgmt.<VTSS_MODULE_ID_DEMO>.11.13.17
```

where `vtss` and `vtssSwitchMgmt` is defined in `VTSS-SMI.mib` which can be found in `vtss_appl/snmp/mibs/`.

If `VTSS_MODULE_ID_DEMO` is defined to `148` in `vtss_appl/include/vtss/appl/module_id.h`, the `OID` of `_demo_ex1` will be `enterprise.6603.1.148.11.13.17`.

[1] There is a similar hook-up for JSON in `demo_json.cxx` which is quite similar, but we will use SNMP as the example for now.

In the actual file the numbers 11, 13 and 17 are 1,1 and 1, but in order to illustrate which number that goes where, I have changed them to the 3 prime numbers above. From now on they will be one's.

The `_demo_ex1` is only the name of this object in the C code. And if you look it is not references from anywhere. The name of this object in SNMP context is `vtssDemoEx1`, which is the second parameter in the `OidElement()` object which is given as parameter in the definition of `_demo_ex1` with a pre-pended `vtss`. The SNMP MIB will wrt this then look like this, where I have left out the parameters in each definition

```
vtssDemoMib MODULE-IDENTITY
::= { vtssSwitchMgmt 148 }

vtssDemoMibObjects OBJECT IDENTIFIER
::= { vtssDemoMib 1 }

vtssDemoStatus OBJECT IDENTIFIER
::= { vtssDemoMibObjects 1 }

vtssDemoEx1 OBJECT IDENTIFIER
::= { vtssDemoStatus 1 }
```

One more important thing to notis is the `demo_mib_init` in `VTSS_MIB_MODULE`. This is the name of the init function which must be called in order to register the MIB. In `demo.cxx` it is shown how it is called.

1.2 Implementation of DemoEx1

The implementation of `DemoEx1` can be found in `demo_serializer.hxx`.

First the type `P` is defined from the template `ParamList`. In our case this template has only one argument namely `ParamVal<demo_ex1_attr_t *>`, but it can have any number[2]. For each argument there must be a `VTSS_EXPOSE_SERIALIZE_ARG_<N>` where `<N>` is the argument in `ParamList` to which the definition apply. In this case we have only `VTSS_EXPOSE_SERIALIZE_ARG_1(demo_ex1_attr_t &i` which expand to

```
template <typename HANDLER>
void argument(HANDLER &h, ::vtss::expose::arg::_1, demo_ex1_attr_t &i)
{
  // BODY of VTSS_EXPOSE_SERIALIZE_ARG_1 which is
  h.argument_properties(vtss::expose::snmp::OidOffset(1));
  serialize(h, i);
}
```

The `h` object is supposed to implement different methods. Above is can be seen, that `argument_properties()` must exist. In this specific case the offset `OID` of the SNMP attribute that is object implement. In the `serialize()` function, which also is located in this file. In this function everything is more or less about doing something to `h`. In order to do this, we must know what `HANDLER` in `argument()` is.

The

[2] Any number means at the most 16.

```
VTSS_EXPOSE_GET_PTR(demo_ex1_get);
```

expand to

```
static constexpr mesa_rc (*get)(demo_ex1_attr_t *) = demo_ex1_get;
```

It can be difficult to follow the implementation to see this, but it should not be necesssary either. A similar thing apply to the set function. These function are made available in the `DemoEx1` class just like the `argument()` method that we looked at earlier. So this is what is made available to the SNMP interface in `demo_mip.cxx` that we started with.

As can be seen, the `argument()` function must be given the value of the attribute and does then specify how it shall be presented. The `get()` and `set()` functions can provide the value.

1.3 Implementation of DemoEx2

This is an example of an attribute, just like DemoEx1, which also is associated with a trap. The attribute is read-only from a management point of view. In this example the value can be changed with an ICLI command, namely

```
# demo ex2 set <number>
```

That will set the value of this attribute to `<number>` and emit a trap if the value has changed. Comparing this example with DemoEx1, it should be obvious what is needed in order to add trap support. As an exercise you could try to do that for DemoEx1.

1.4 Implementation of DemoEx3

This is an example of a table with two values per table entry and indexed with an integer.

1.5 Implementation of DemoEx4

This is an example is similar to DemoEx3, but now the index is `ifIndex`.

1.6 Implementation of DemoEx5

This is an example is similar to DemoEx3, but now the index is a 3 dimentional. So this illustrate how iterators over many keys are aggregated.

1.7 Implementation of DemoEx6

This is an example of a table into which you can add, delete and modify values. The object `demo_ex6_status` is registered with this table in `demo_mib.cxx`. Initially the table is empty, but values can be added or modified with the objects `set()` methos. And table rows can be deleted with the 2codedel() metode. From the function `demo_ex6_generate_trap()` in `demo_attribute.cxx` it should be easy to see how this works.

This function is called from ICLI, so doing

```
# demo ex6 set 3 21
# demo ex6 set 3 22
# demo ex6 del 3
```

will create a row with index 3 and value 21 in attr1. This will also emit a add trap. The second command will change the value of the newly create row and emit a modification trap. Finally the last command will delete the row and emit a delete trap.

1.8 Implementation of DemoEx7

This is an example of a read/write/add/delete table. The definition of DemoEx7 in demo_serializer.hxx shows, that there are a function pointers for add, del and def in addition to the get, set and itr function we have seen i earlier examples

In demo_mib.cxx _demo_ex7 is defined, and it has an OID for the table and one for a table row editor. In SNMP the table looks like this

```
status -+- Table(70) -- Entry -+- index(1)
        |                      +- attr1(2)
        |                      +- attr2(3)
        |                      +- action(123)
        |
        +- TableRowEditor(71) -+- index(1)
                               +- attr1(2)
                               +- attr2(3)
                               +- action(123)
```

The actual table is under Table. As can be seen, there is an extra element, namely action. This action is of type VTSSRowEditorState and is defined in VTSS-TC.mib where a description of the functionality of this field can be found too.

1.8.1 Specifying the OID

The OID of the diffenrent elements are specified in 3 places. The 70 and 71 are specified in the definition of _demo_ex7 in demo_mib.cxx.

The 1, 2 and 3 for the attributes are in demo_serializer.hxx. For example the OID of attr1 and attr2 are specified in VTSS_EXPOSE_SERIALIZE_ARG_2() in the definition of DemoEx7 to begin at 2 by the OidOffset(2). In the associated serialize() function it can be seen, that attr1 is specified with OidElementValue(0) and attr2 with OidElementValue(1). So for attr1 the OID is 2+0 and for attr2 it is 2+1.

Finally the action OID is 123, which is specified in the definition of DemoEx7 by the snmpRowEditorOid.

1.8.2 The action attribute

The action attribute appear two places, namely in the table itself and in the row editor

For the action in the table: When reading this action attribute, the value is always 0. If something different for 0 is written to this field, then the row will be deleted. E.g. the command below will delete row with index 2[3]. The function demo_ex7_del() is called with index=2.

```
$ snmpset -c private -v 2c <IP> 1.2.6.1.4.1.6603.1.148.1.1.70.1.123.2 u 1
```

For the action in the row editor: In order to create a row, the TableRowEditor is used. The action at this point works as two things - a semaphore and as a command register. This interface is described in VTSS-TC.mib. In short a number greater than 255 must first be written to this attribute. There can potentially be many managers, and each manager should use a different ID when wanting to modify an entry. After writing an ID, you should

[3] i.e. the 3rd row since we count from 0

read it back. If you get your number back, then the row editor is yours and you can write to the other attributes - in our case `index`, `attr1` and `attr2`. When you are done write COMMIT-ACTION, i.e. 2 to the `action` attribute. The `demo_add_ex7()` function is called and after that the `demo_def_ex7()` to default the attributes in the row editor again.

If you instead of COMMIT-ACTION(2) write CLEAR-ACTION(1), then only the `demo_def_ex7()` function is called. If you write IDLE(0), then you will release your semaphore, but changed to the row editor will not be used or changed. So if managers expect the row editor to have the default values when they allocate this resource, then you should not do that.

1.9 JSON interface

The JSON interface is similar to SNMP. Try to compare `demo_mip.cxx` and `demo_json.cxx`.

It is recommended to use the `vson` tool to test JSON object. It can be donwloaded for git

```
$ git clone https://github.com/vtss/json-rpc-util.git
```

The first thing to do when using this tool is to download the JSON specification from the target. This is done with[4]

```
$ ./vson -d <IP> -c update-spec
```

which will create the file `.vtss-json-rpc.spec` in the root of your home directory. In order to find the methods for this demo project you can say

```
$ ./vson -d <IP> -c grep demo
```

and you'll get something like this

```
demo.something.simple_ex1.get
demo.something.simple_ex1.set
demo.something.notif_ex2.get
demo.something.table_ex3.get
demo.something.table_ex3.get
demo.something.table_ex3.set
demo.something.table_ex4.get
demo.something.table_ex4.get
demo.something.table_ex4.set
demo.something.table_ex5.get
demo.something.table_ex5.get
demo.something.table_ex5.set
demo.something.table_ex6.get
demo.something.table_ex6.get
demo.something.table_ex7.get
demo.something.table_ex7.get
demo.something.table_ex7.set
demo.something.table_ex7.add
demo.something.table_ex7.del
```

There is at least one get method for each object, and one of these do not take any argument. For Ex1 there is only one get method which do not take any argument.

[4] The <IP> is the IP address of the target.

```
$ ./vson  -d <IP>  -c call demo.something.simple_ex1.get
Calling demo.something.simple_ex1.get:
    Attr1: false
    Attr2:     0
```

For tables there are two get methods; one that does not take any argument and will get the entire table, and another that take the table index as a parameter and will get you that row.

In order to figure out how to set the attribute, it is suggested that you dump the json request in the above get command, i.e.

```
$ ./vson  -d <IP>  -c call --dump-response demo.something.simple_ex1.get
Calling demo.something.simple_ex1.get:
  Post-body: {"id":"jsonrpc","error":null,"result":{"Attr1":false,"Attr2":0}}
    Attr1: false
    Attr2:     0
```

The result object reveale howto set this attrhbute

```
$ ./vson  -d <IP> -c call  demo.something.simple_ex1.set \
                             '{"Attr1":true,"Attr2":34}'
{"Attr1":true,"Attr2":34}
Calling demo.something.simple_ex1.set:
```

In order to get table Ex5 say:

```
$ ./vson  -d <IP>  -c call demo.something.table_ex5.get
Calling demo.something.table_ex5.get:
    key       Attr1_Ex5 Attr2_Ex5
    --------- --------- ---------
    [0, 0, 0]         0         0
    [0, 0, 1]         0         1
   ...
    [5, 5, 4]        10         9
    [5, 5, 5]        10        10
```

In order to get one row say

```
$ ./vson  -d <IP>  -c call demo.something.table_ex5.get 1 2 3
```

which will use the other get method available for this example. A set operation looks like this

```
$ ./vson  -d <IP>  -c call demo.something.table_ex5.set 1 2 3 \
                              '{"Attr1_Ex5":4,"Attr2_Ex5":5}'
```

In Ex4, the `ifIndex` is used as an index. In JSON this is a string:

```
$ ./vson  -d <IP>  -c call demo.something.table_ex4.get "Gi 1/1"
```

1.10 JSON notification

Just like we have SNMP traps, we have JSON notifications. In order for a notification to be sent, it must be enabled. In order to enable notifications for Ex2 give the following ICLI commands:

```
# configure terminal
(config)# json notification host mydemo
(config-json-noti-host)# url http://a.b.c.d:5000
(config-json-noti-host)# exit
(config)# json notification listen demo.something.notif_ex2.update mydemo
```

and then on machine with IP address `a.b.c.d` start the listner:

```
$ ./listen -p 5000
```

Now you can test it by changing the Ex2 attribute as described earlier.