



# **sat-nms Software User Manual**

## **Universal Device Driver Reference Manual**

Version 3.80.6-a -- 2025-12-03

© Copyright  
SatService Gesellschaft für Kommunikationssysteme mbH  
Hardstrasse 9  
D-78256 Steisslingen

[satnms-support@satservicegmbh.de](mailto:satnms-support@satservicegmbh.de)

[www.satnms.com](http://www.satnms.com)  
[www.satservicegmbh.de](http://www.satservicegmbh.de)  
Tel +49 7738 99791-10

## Table of Contents

Table of Contents	2
1 Device Driver Reference	6
1.1 Understanding the device driver	7
1.1.1 Variables	8
1.1.2 Procedures	9
1.1.3 Protocol encapsulation	9
1.2 Writing a device driver	10
1.2.1 Starting from scratch	10
1.2.1.1 General file format	11
1.2.1.2 Naming a device driver	11
1.2.1.3 Selecting a communication protocol	12
1.2.1.4 File includes	13
1.2.1.5 Tweaking the processing sequence	14
1.2.2 Defining variables	14
1.2.2.1 The VAR statement	15
1.2.2.2 The ALARM statement	20
1.2.2.3 Info variables	21
1.2.2.4 Setup variables	22
1.2.2.5 Variables for internal use	23
1.2.3 Using conversion tables	23
1.2.3.1 The TABLE statement	24
1.2.4 Adding data exchange procedures	24
1.2.4.1 The PROC statement	25
1.2.4.2 ASYNC procedures	27
1.2.5 Basic I/O functions	29
1.2.5.1 The PRINT statement	30
1.2.5.2 The INPUT statement	33
1.2.5.3 The WRITE statement	37
1.2.5.4 The READ statement	41
1.2.5.5 Function tables in I/O functions	45
1.2.6 REST I/O functions	46
1.2.6.1 The REST TRANSACT statement	49
1.2.6.2 The REST PARSE statement	50
1.2.6.3 The REST SET statement	52
1.2.6.4 The REST CLEAR statement	56
1.2.6.5 The REST SEND statement	56
1.2.7 Conditional execution	57
1.2.7.1 The IF statement	57
1.2.7.2 The GOTO statement	57
1.2.7.3 Label definition	58
1.2.8 Using subroutines	58
1.2.8.1 The CALL statement	59
1.2.9 Manipulating variables	59
1.2.9.1 The SET statement	60
1.2.9.2 The BITSET statement	60
1.2.9.3 The RANGESET statement	61
1.2.9.4 The BITSPLIT statement	65

1.2.9.5 The BITMERGE statement	65
1.2.9.6 The SEND statement	65
1.2.9.7 The COPY statement	66
1.2.10 More statements	66
1.2.10.1 The DELAY statement	67
1.2.10.2 The LOG statement	67
1.2.10.3 The DRATE statement	68
1.2.10.4 The SRATE statement	69
1.2.10.5 The WRITEHEX statement	70
1.2.10.6 The READHEX statement	71
1.2.10.7 The INVALIDATE statement	71
1.2.10.8 The SYNC statement	71
1.2.10.9 The RAISECOMMFAULT statement	72
1.3 The RPN language extension	72
1.3.1 The RPN stack	73
1.3.2 The { ... } statement	74
1.3.3 RPN command reference	75
1.4 SNMP device classification and variable binding	80
1.4.1 Device driver syntax for SNMP definitions	80
1.4.2 List of SNMP device classes and OID names	82
1.5 Device driver examples	85
1.5.1 NDSatCom-KuBand-Upconverter Example	85
1.5.2 Tandberg-Alteia Example	87
1.6 Device communication protocols	101
1.6.1 Writing a communication protocol definition	102
1.6.2 General file format	103
1.6.3 Global definitions	104
1.6.4 TX message elements	105
1.6.4.1 CHAR	106
1.6.4.2 ADDRESS	106
1.6.4.3 USERDATA	106
1.6.4.4 CHECKSUM	107
1.6.4.5 DATALENGTH	108
1.6.4.6 HEXLENGTH	108
1.6.4.7 SEQUENCE	108
1.6.5 RX message elements	109
1.6.5.1 START	109
1.6.5.2 CHAR	109
1.6.5.3 ADDRESS	110
1.6.5.4 USERDATA	110
1.6.5.5 STRING	111
1.6.5.6 CHECKSUM	111
1.6.5.7 DATALENGTH	113
1.6.5.8 HEXLENGTH	113
1.7 Device oriented user interface	113
1.7.1 How the software finds the screens for a device	114
1.7.2 Creating new screens	114
1.7.3 Creating a 'frame' definition	115
1.7.4 Icon reference	116

1.8 Online Help	119
1.8.1 Help file format	119
1.8.2 Rebuilding the online help	123
1.8.3 Adding help files for new devices	124
1.9 ServiceClient library	124
1.9.1 Adding new devices or device types to the library	125
1.9.2 Adapting the appearance of the ServiceClient	125
1.9.3 File format of definitions.xml	125
1.9.4 Device type definitions	125
1.9.4.1 Parameters	125
1.9.4.2 Implementations	125
1.9.5 GUI constants	125
1.10 Reference Tables	128
1.10.1 Device driver keyword reference	128
1.10.2 Protocol definition keyword reference	132
1.10.3 Help file keyword reference	133
1.10.4 Debugging with Terminal Session	134
1.10.4.1 Terminal Session Commands	134
1.10.4.2 Device Message Commands	139
1.10.5 Global faults	141



# 1 Device Driver Reference

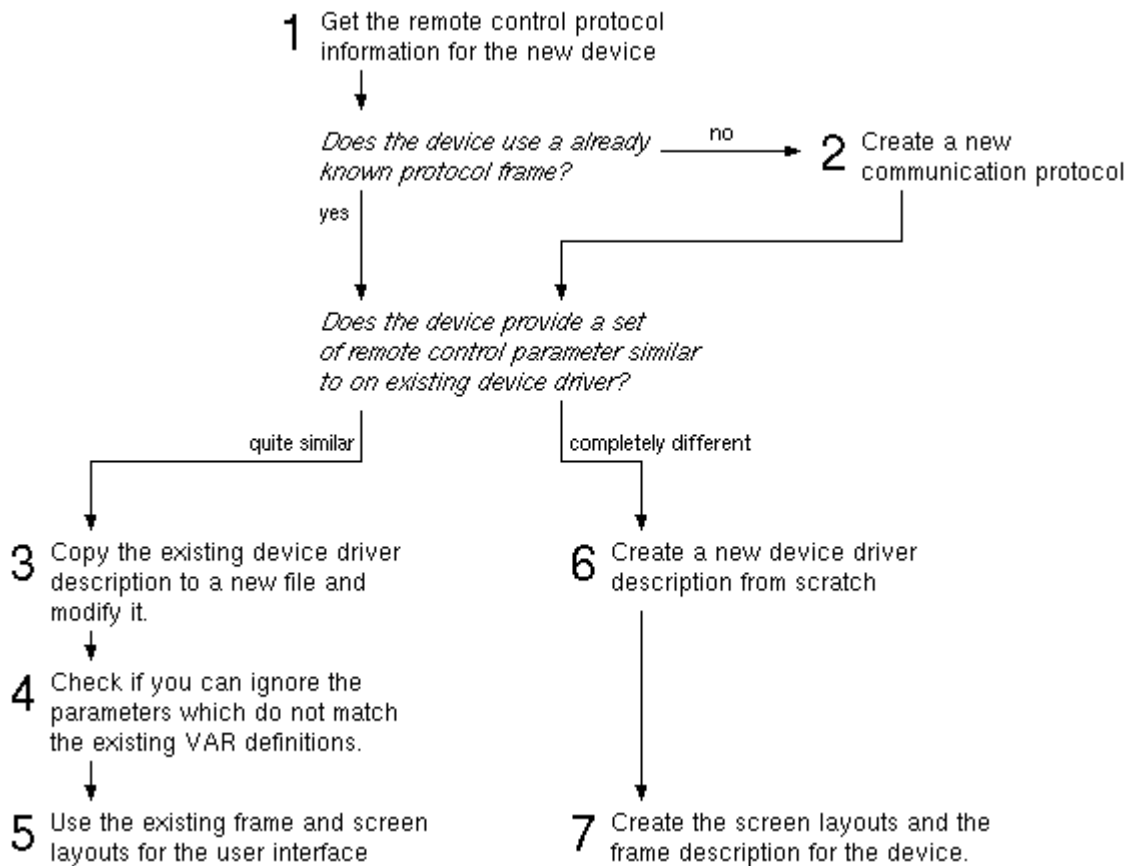
Device drivers in the MNC software translate abstract parameter settings which are represented by parameter messages into commands sent to the physical device and vice versa. The basic idea is to modularize the software in a way, that one device in a station setup can be replaced by another model, perhaps even by a model made by another vendor, simply by selecting another device driver.

The **sat-nms** MNC advances this concept by introducing a **universal device driver** which is completely user configurable. The configurable device driver let's you write your own device drivers for device models which are not yet supported by the software. Most of the device drivers coming with the software are built on top of this configurable driver, so there are a lot of examples you can use as a template.

To make the **sat-nms** software support an additional device type, the software needs at least three components:

1. A low level [communication protocol](#) which handles the protocol frame including device addressing or checksum calculation.
2. A [device driver](#) which defines the parameters the operator may inspect or control at the device. The device driver also contains the I/O routines to exchange these parameters with the device.
3. Finally, there must be a standard user interface (the so called [device oriented user interface](#)) for the new device which lets an operator view or modify the device parameters.

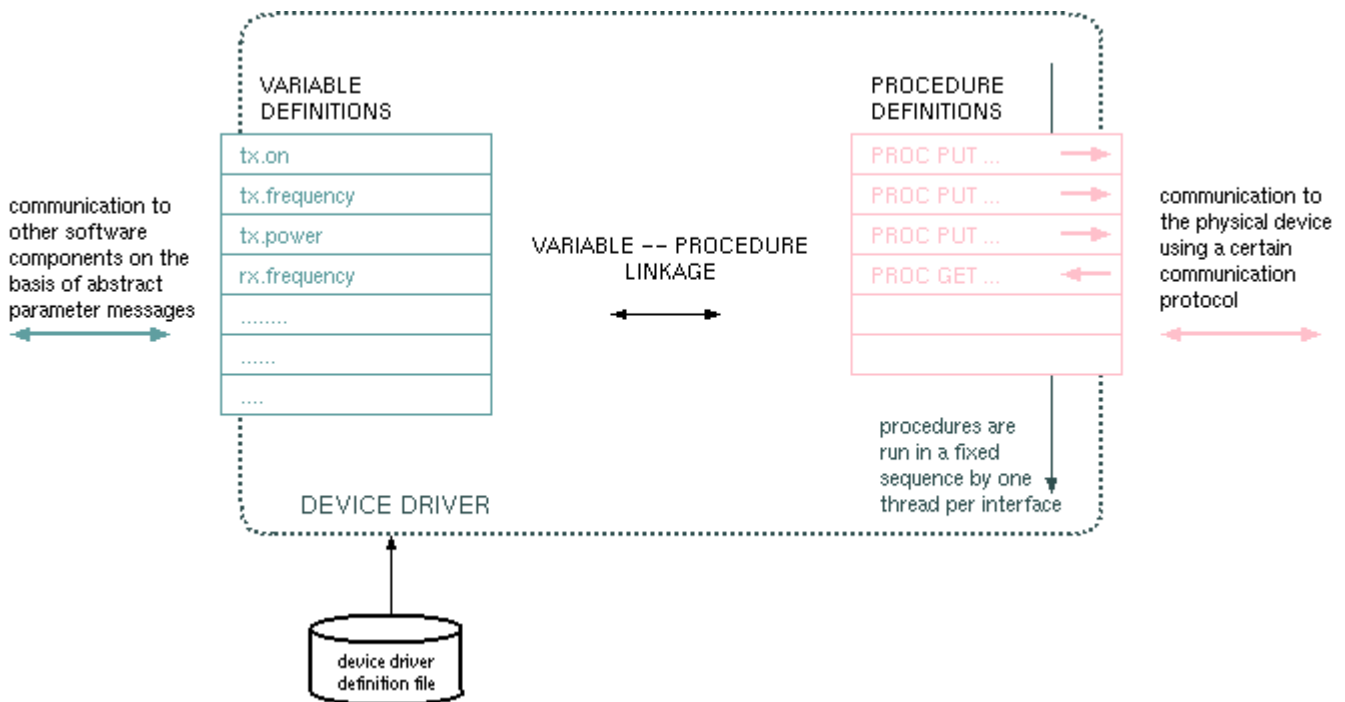
With the **sat-nms** software, all three components from the physical M&C interface at the device up to the representation of parameters on the screen are configurable and extensible by the customer. The **sat-nms** software comes with library supporting a large number of devices. These files are at your's disposal to use them as templates or as ready made module for the device driver you need to develop.



The diagram above illustrates the first steps to the integration of a new device type into the **sat-nms** software. You are encouraged to use as much as possible of the existing protocol, driver and user interface definitions.

## 1.1 Understanding the device driver

The figure below illustrates the structure of a device driver built with the **universal device driver**. The principal function, however, applies to 'hard coded' drivers, too.



To other software components (primarily to the user interface) the device driver interfaces by a list of variable definitions. On the other hand, a list of procedures does the 'real work', translating the abstract parameter values to physical command sequences. In between a function here called *variable -- procedure linkage* defines which procedure is run to set or read a certain parameter.

With the universal device driver, all this is setup (compiled) during the program initialization from a text file which describes the device driver in special, but quite simple programming language. The following pages describe how [variables](#), [procedures](#) and the [protocol encapsulation](#) work together.

### 1.1.1 Variables

Variables are the interface between the device driver and the other components of the software, specially the user interface. Each variable acts as an end point for a parameter message. It may receive messages -- which causes the driver to set this parameter at the physical device -- but also may send it's parameter message in order to tell the user interface about the actual setting of this parameter.

Inside, a variable separately stores two values:

- The value which recently has been commanded and
- the value which has been read from the device during the previous polling cycle.

By managing these values separately, the device driver is able to check if a value has properly been set by the device. If you see messages like `Parameter some.name set to X but reads Y` in the event log, then the parameter polling which followed a command returned a value different from the commanded one.



### 1.1.2 Procedures

Driver procedures actually perform the communication with the device to control. They operate the device as commanded and poll the equipment settings and state.

There are two types of procedures a driver may define.

- A **PUT** procedure sends one or more parameters to the physical device after coding them into the format the physical device expects.
- A **GET** procedure sends a request to the device, reads the reply and decodes one or more variable values from the data received.

A procedure never can be both, **PUT** and **GET** at the same time.

The device driver executes the defined procedures in an endless loop, called the polling cycle. But a procedure is not necessarily called in every cycle. If there is nothing to do for a particular procedure, it is skipped. To determine which procedure must be executed in a cycle, procedures are bound to variables. A procedure may be bound to one or more variables, however, there may be at most one **PUT** and one **GET** procedure referencing a variable.

A **PUT** procedure is executed, if at least one of the variables it is bound to has received a new setting, e.g. from the user interface. For a **GET** procedure the following conditions cause the procedure to be executed:

1. The device driver has established communication to the device after power-on or after a communication interruption.
2. The (individually defined) polling interval for at least one of the variables bound to this procedure has elapsed.
3. At least one of the variables bound to the procedure has been commanded to the device. The parameter must be read back for verification.

### 1.1.3 Protocol encapsulation

The **sat-nms** device driver concept encapsulates the protocol frame used for the communication with a device in a separate layer. This protocol layer describes which kind of protocol frame has to be wrapped around the commands sent to a device. There are a number of important advantages with this concept:

- The protocol frame is automatically added to each command or request sent to the device. The other way round, the driver automatically strips of this 'envelope' from the data received from the device.
- A certain type of protocol frame needs to be programmed only once and may be used for several device drivers, if for example different devices from the same vendor use one and the same protocol definition.
- Some devices may be operated either by one or the other communication protocol. The protocol encapsulation allows easily to switch between the variants.

With the **sat-nms** software, new protocol definitions may be added to the software, simply by editing a text file for the new type of protocol frame which is needed.

If you are going to write a new device driver, you first should investigate if there is an existing protocol definition in the **sat-nms** library which can be used to serve the new device. If no protocol definition matches, the chapter [Device communication protocols](#) describes to write a new protocol definition.

## 1.2 Writing a device driver

As mentioned above, device drivers are coded as simple text files. When the MNC program starts, it 'compiles' the device drivers needed for it's equipment setup to memory. Writing a device driver means editing such a text file and storing it at a place in the MNC computer where the MNC program searches for device drivers.

The **sat-nms** device driver language has been designed to be very easy to understand. If you have a look at the device driver files coming with the **sat-nms** software, you probably will understand most of this language after an hour or two.

A MNC or VLC system keeps all device drivers in a subdirectory called `drivers`. On standard installations this is the directory `/home/satnms/drivers`. The name of the device driver files consist of the driver name (usually something like `Manufacturer-ModelNumber`) followed by the extension `.device`.

If you are writing the device driver on a Linux based MNC or NMS computer, you may want to use the VI text editor or mcedit (included in midnight commander) text editor for this. Both has been configured to colorize device driver files on these machines. We also providing syntax definitions for Visual Studio Code. Please ask your contact at SatService for details.

You also may copy device driver files to a MS-Windows based computer to edit them there. If you do this, you should consider the following:

- Device driver files are Unix based text files. Lines are terminated with a line-feed character only. Your favorite MS-Windows text editor may have problems to show these files.
- Unix / Linux is case sensitive with file names. Be sure that you don't mess up the case of characters in file name when you copy files between Unix and MS-Windows.

### 1.2.1 Starting from scratch

The following pages describe the 'hard way' to create a device driver from scratch. It is a good exercise to do this once for a simple driver. In practice however, in most cases you will use an existing driver as a template and modify this for your needs.

The basic steps to build a device driver from scratch are:

1. [Naming the device driver](#)
2. [Selecting the communication protocol](#)
3. [Including the standard definitions file](#)
4. [Defining the driver variables](#)
5. [Adding the data exchange procedures](#)

If you are reading this manual online, you may open the [NDSatCom-KuBand-Upconverter](#) driver example in a separate window to watch this in parallel while you are reading the following pages.

### 1.2.1.1 General file format

The **sat-nms** device driver language has been designed to be very easy to understand. The syntax resembles the BASIC programming language in some aspects, however, the language is highly specialized for it's purpose. Here some basic definitions which help you to read or write device driver files:

- The device driver language is case sensitive for all identifiers and keywords.
- Whitespace (space characters, tabs, line breaks) separates words.
- Line breaks have no special termination function.
- All keywords (except the RPN commands) are in upper case letters.
- Comments in C/C++ style are recognized (both, `/* ... */` and `// ...` comments).
- Identifiers (names for variables and tables) may consist of letters, digits and dots. They must start with a letter.

Beside the rules listed above, the device drivers provided by SatService GmbH follow some sort of coding/formatting convention:

- Each file starts with a comment block containing a short description and a change history.
- Identifiers always start with a lowercase letter. If a name consists of more than one word, the first letter of each following word is in upper case. Dots are used to group parameter functionally. Example: `mod.symbolRate` .
- Statements are indented four columns for each level.

### 1.2.1.2 Naming a device driver

To create a new device driver, the first step is to give the driver a name. The device drivers supplied by SatService GmbH all follow a 2 part naming scheme:

`VendorName-ModelDescription` . A SatService ACU2 antenna control unit appears as `SatService-ACU2` in the device driver list. This makes it easy for the person who configures the equipment setup of a MNC system to select the appropriate device drivers.

The name of the device driver appears at several places:

- It is used as part of the device driver file name.
- Exactly the same name should be defined as a initialized variable called `info.type` . The online help function and the device preset directory selection of the user interface depend on this definition. Chapter [Info variables](#) gives more information about this.
- The COMMENT statement also contains the device driver name.

### The COMMENT statement

**COMMENT** → " *text* "

The text following the COMMENT statement is free field and principally may contain any information. The device drivers coming with the **sat-nms** software all follow a convention which defines the comment string as

### Driver-name X.YY YYMMDD

where X is the major version number of the driver, YY the minor version number and YYMMDD the release date of this driver version. It is recommended that customer defined device driver follow this scheme, too.

The comment statement in fact creates a hidden info-variable definition. This variable with the name `info.driver` is initialized with the comment text. At the Info-page of the device oriented user interface you can view the comment text or the driver name /version / date respectively.

### Example

```
COMMENT "NDSatCom-KuBand-Upconverter 1.03 010809"
```

This example, taken from NDSatCom-KuBand-Upconverter device driver, identifies this driver as version 1.03, released at August, 9th, 2001.

### 1.2.1.3 Selecting a communication protocol

With the **sat-nms** software, the device communication protocol handles the low level communication between MNC and device. There are trivial protocols which simply put a newline character at the end of each message and more sophisticated ones dealing with start/end characters, checksums, device addresses and more.

The **sat-nms** software comes with a bunch of predefined protocol definitions. If none of these protocol definitions matches, the chapter [Device communication protocols](#) describes to write your own protocol definition.

The communication protocol used for a certain device (for a certain serial interface, more exactly spoken) is defined with the equipment setup which you can configure at the graphical user interface. The device driver defines a preferred communication protocol. This means, in the device driver you state the communication protocol this driver is designed for. At the equipment setup screen of the software you still are able to choose a different protocol for a device of this type, however, you will be warned by the software. The preferred communication protocol for a device driver is defined with the PROTOCOL statement:

**PROTOCOL** → *protocol-name*

The protocol definition this statement refers to must exist in the `protocols` subdirectory of the software. On standard installations this is the directory `/home/satnms/drivers`.

### Example

PROTOCOL Miteq-MOD95

This example, taken from NDSatCom-KuBand-Upconverter device driver, defined the Miteq-MOD95 protocol as the preferred one for the driver.

Some communication protocols may be configured in some aspects of their functionality by applying protocol parameters to them. This is done with the `PROTOCOLPARAMETER` statement:

**PROTOCOLPARAMETER** → " *key=value* "

There may multiple `PROTOCOLPARAMETER` statements in on device driver file, each defining one aspect of the communication protocol. The `key` defines the parameter to set, `value` is the value to be assigned to this parameter. Please refer to the documentation of the protocol you are using (usually given in the comment section of the `*.proto` file) if there are configurable parameters for this protocol and if you need to set them for your application.

Protocol parameters may be set either in the protocol definition file (thus defining the parameter for all uses of the protocol) or in the device driver. A definition in the device driver overwrites the setting in the protocol file.

**ATTENTION:** If you have multiple devices in the same device thread (`INTERFACE`) using different device drivers, the result is unpredictable if the drivers do not make exactly the same protocol parameter settings.

### Example

`PROTOCOLPARAMETER "post.content.type=application/xml"`

#### 1.2.1.4 File includes

**ATTENTION:** File including does not work when a device driver file gets interpreted by the *client* software, e.g. when a device preset shall be formatted along the device driver's parameter definitions. Variables defined in included files get not properly formatted in the device preset window. **For this reason it is recommended to use `INCLUDE` only for the standard includes as described in the example below!**

Device driver definition files may include other files. Frequently needed definitions may be encapsulated in included files which makes it easy to change these definitions at a central point. The syntax of the `INCLUDE` statement is:

**INCLUDE** → " *file-name* "

The file name given in the `INCLUDE` statement is relative to the MNC software's home directory. With the **sat-nms** Software, there are two standard include files. Almost any driver provided by SatService GmbH includes one of these files. The files are named `Standard.inc` and

`StandardBin.inc` respectively. They define the so called low level interface to the device and the fault flags which indicate a communication failure for the device.

### Example

```
INCLUDE "drivers/Standard.dotinc"
```

This includes the standard ***sat-nms*** include file for driver which use a text based communication protocol. The file is located in the `drivers` subdirectory which is one level below the MNC software's home directory.

#### 1.2.1.5 Tweaking the processing sequence

By default, the device driver executes the `PUT` and `GET` procedures in the same sequence, the procedures appear in the the driver file. Procedures which are not ready to run are skipped, but the overall sequence of execution is given by the order in which the procedure appear in the driver.

For `PUT` procedures this has the implication, that they are not executed immediately after an operator (or commanding device) has sent a new value for a parameter. The procedure has to wait it's turn. If there are a lot of `GET` procedures ready to run, this may take some time, specially with complex devices which are quite slow in response.

By specifying the `PUTPRIORITY` mode in the header of the device file, you can speed this up, making the device driver for a slow device more responsive:

### PUTPRIORITY

You enable the `PUTPRIORITY` mode for a driver by adding `PUTPRIORITY` as a top level keyword to the driver file. This changes the order of execution of procedures in the following way:

If at least one `PUT` procedure of the driver is ready to run, all `GET` procedures are skipped. `GET` procedures only are executed if no `PUT` procedure is ready to run. The outcome of this is, that the `PUT` procedure is executed not later than the device thread's `IDLE` time after the new value has been commanded.

But as with most good things there is one hitch with this: If you change a parameter once every driver cycle of faster, `GET` procedures will never get a chance to run. The driver will respond quickly and execute any settings you command, but state information like alarms or meter readings will never be updated. This is the reason why `PUTPRIORITY` is not the default behavior of a ***sat-nms*** device driver.

#### 1.2.2 Defining variables

Variables are the interface between the device driver and the other components of the software, specially the user interface. Each variable acts as an end point for a parameter message. It may receive messages - which causes the driver to set this parameter at the

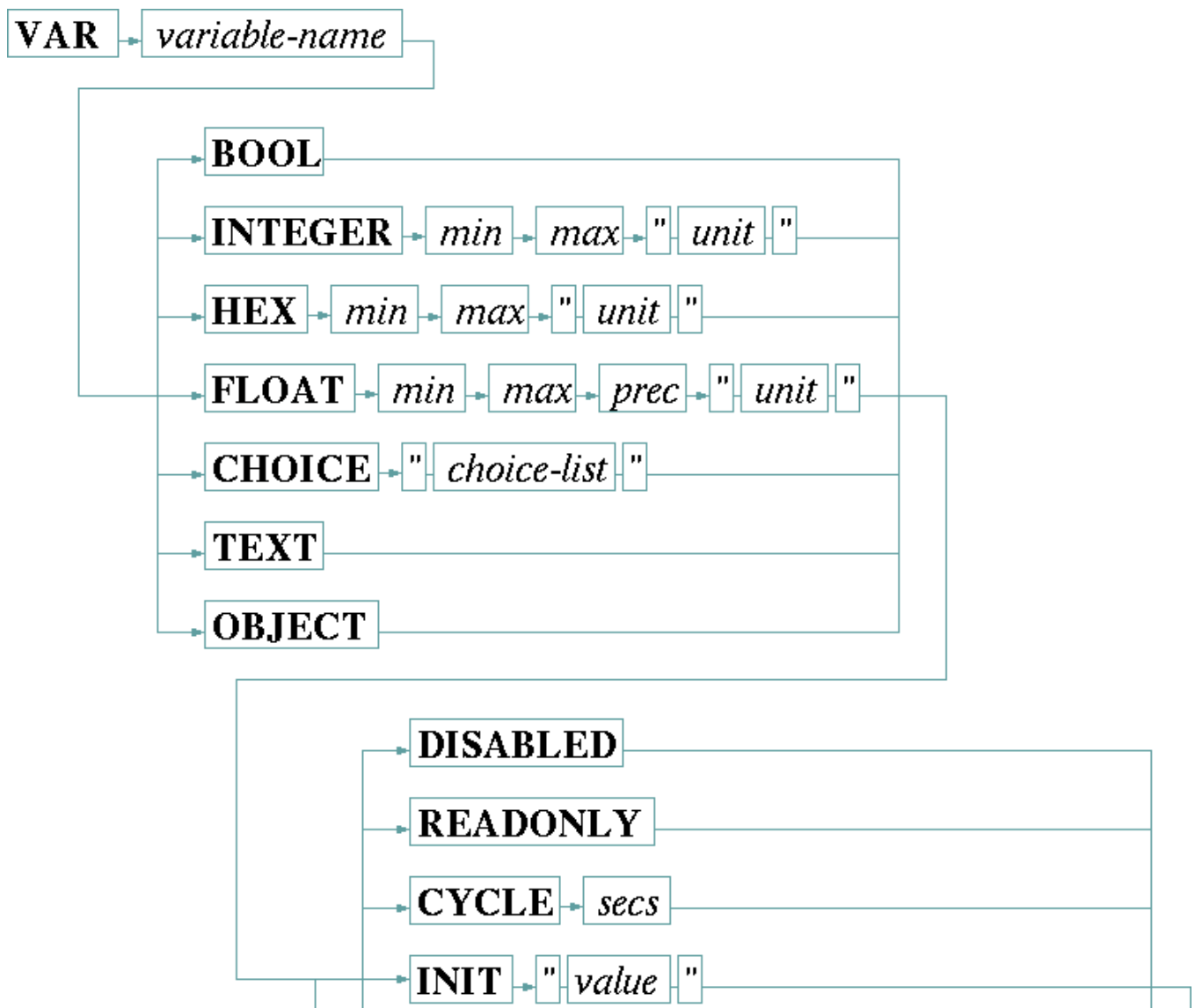
physical device - but also may send it's parameter message in order to tell the user interface about the actual setting of this parameter.

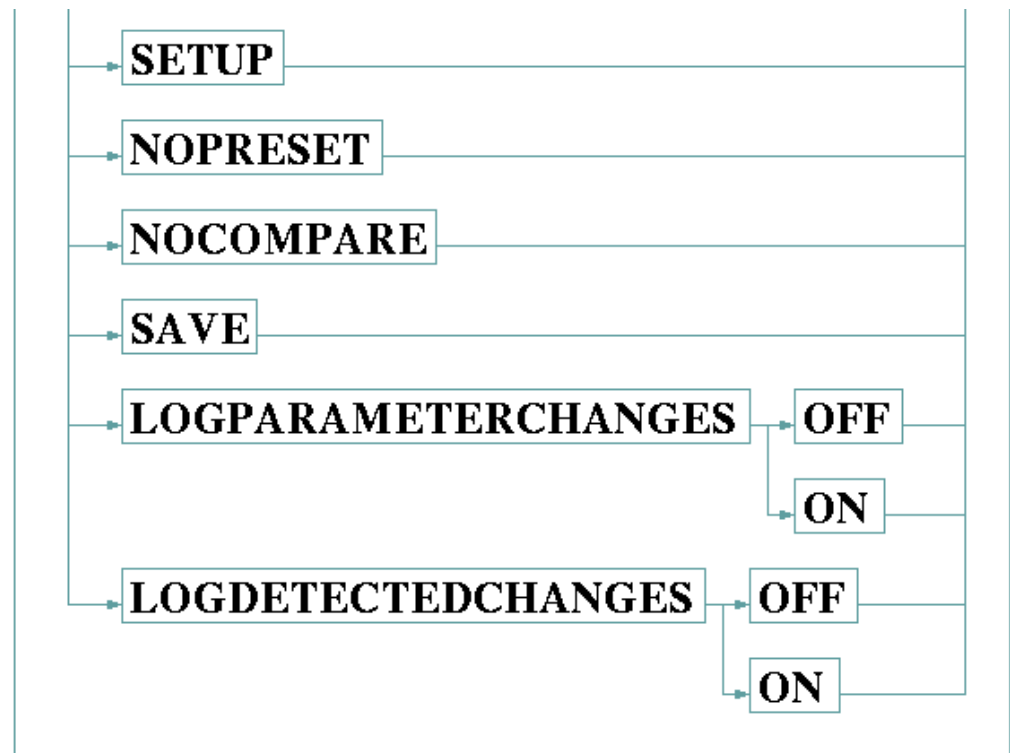
Before you start writing the device driver, you should plan a list of variables which build the software interface to the device. Designing this interface should be done careful, you can save a lot of effort if you are able to re-use variable definitions from other devices. This not only saves the time to write the definitions, it also enables you to use the parameter screens available for this device in the **sat-nms** library.

If you intend to write are driver for a device which implements similar functions as an existing driver, it is recommended to copy the variable list from this device.

### 1.2.2.1 The VAR statement

Device driver variables are defined using the **VAR** statement. With the **VAR** statement you define the name, the data type, the valid range and some other properties of the variable. This is the general syntax for a VAR statement:





A variable definition consists of the VAR keyword, the variable's name, a data type / range definition and optional modifiers which control the behavior of this variable.

### Data type / range definition

The **sat-nms** device driver knows about 7 data types. The data type of a variable is defined by one of the data type keywords `BOOL`, `INTEGER`, `HEX`, `FLOAT`, `CHOICE`, `TEXT` or `OBJECT`, followed by the range definition for the selected data type. Here the data types listed in tabular form:

<b>BOOL</b>	The <code>BOOL</code> data type can have the values <code>true</code> and <code>false</code> . the <code>BOOL</code> type mainly is used for flags which shall be displayed as signal lamps at the user interface.
<b>INTEGER</b>	The <code>INTEGER</code> data type carries numeric integer values (64 bit length). When you define an <code>INTEGER</code> variable, you must provide the valid range ( <code>min</code> / <code>max</code> ) of the variable and a unit string which is shown at the user interface right of the data. If both, <code>min</code> and <code>max</code> are zero the user interface software does no range check at all. The unit string may be empty, but the double quotes are required ("").
<b>HEX</b>	The <code>HEX</code> data type also is a 64 bit integer, but formatted in hexadecimal notation.



<b>FLOAT</b>	<p>The <code>FLOAT</code> data type carries double precision (64 bit length) floating point values. When you define a <code>FLOAT</code> variable, you must provide the valid range ( <code>min</code> / <code>max</code> ) of the variable, the number of fraction digits and a unit string which is shown at the user interface right of the data. If both, <code>min</code> and <code>max</code> are zero the user interface software does no range check at all. The unit string may be empty, but the double quotes are required ("").</p> <p><code>FLOAT</code> variables are displayed with a fixed precision. Alternatively you may force a scientific notation by adding 100 to the precision value. Example: <code>VAR myFloat FLOAT 0 0 103 ""</code> defines a scientific formatted floating point variable shown with 3 digits precision (e.g. <code>0.123E-2</code> )</p>
<b>CHOICE</b>	<p>The <code>CHOICE</code> data type defines a parameter which may contain one of a fixed set of string values. You define this set as a comma separated list, enclosed in double quotes. At the user interface such a parameter appears as a drop down box where you can select a value from this list.</p>
<b>TEXT</b>	<p>The <code>TEXT</code> data type carries an arbitrary character string.</p>
<b>OBJECT</b>	<p>The <code>OBJECT</code> data type is used with complex variables which cannot be shown by the standard user interface routines. <code>OBJECT</code> variables only appear together with logical devices, you never will need this data type when writing a device driver.</p>

## Modifiers

Modifiers control some properties of a variable concerning it's state, usage, initialization and polling cycle. Modifiers may be before or after the type definition.

<b>DISABLED</b>	<p>Variables may be <i>enabled</i> or <i>disabled</i>. Disabled variables appear grey at the user interface, no value is displayed and no value may be entered. The <code>RANGESET</code> command is used to change the enable state. By default variables start in enabled state unless they are marked as <code>DISABLED</code> in the VAR statement. Disabling a variable at this point may be useful if the variable stands for a parameter which is not available at all models of the device type the driver is written for. The variable can be enabled by the driver when it detects that the device actually connected to the MNC supports this parameter.</p>
-----------------	---

<b>READONLY</b>	Marking a variable <b>READONLY</b> prohibits the operator from changing it's value. This is used for state variables like meter readings. The range information for the data type used must be provided even if the variable is marked <b>READONLY</b> .
<b>CYCLE</b>	The <b>CYCLE</b> modifier controls the frequency (expressed as time interval in seconds), this variable shall be polled from the device. By default, variables are read from the device with every working cycle of the driver. This is about one a second if only a few parameters are to read. With many parameters the polling rate will be lower as the response time for each interrogation extends the cycle time. Hence, most device drivers poll only a few parameters like alarm flags or some meter readings with the maximum possible rate. Other parameters, e.g. settings you do not expect to change by themselves, are polled at a much lower rate. Setting the <b>CYCLE</b> time to zero causes the driver to do no regular polling for this variable at all. However, after power up and after communication failures, such a variable still is read once from the device
<b>INIT</b>	Using the <b>INIT</b> modifier, a variable may be initialized to a certain value. This value remains valid until the variable gets polled the first time. The <b>INIT</b> option often is used with variables which are used to configure the driver and never are read from the device itself.
<b>SETUP</b>	The <b>SETUP</b> modifier marks a variable to be listed in the maintenance/setup window of the standard device screens. Chapter <a href="#">Setup variables</a> tells more about this special variable type.

<b>NOPRESET</b>	<p>Using the <code>NOPRESET</code> modifier, a variable can be explicitly excluded from the set of parameters which are written to a device preset. A device preset contains all variables which define a <code>PUT</code> procedure and are no <code>SETUP</code> parameters and don't have the <code>NOPRESET</code> modifier set. The <code>reset</code> variable - if defined - is also implicitly excluded from device presets preset, for backward compatibility reasons. Please note, if you add <code>NOPRESET</code> to a variable in an existing device driver, this will not remove the value for this variable from existing device presets, it only will prevent the software from adding the value to new presets. You may use the preset editor to remove the unwanted value manually from existing device presets.</p>
<b>NOCOMPARE</b>	<p>Without the <code>NOCOMPARE</code> modifier the device driver will compare a value it commanded to the device against the value it read back after this. If the two values differ, an informational message showing the commanded and the read value is added to the log. <code>NOCOMPARE</code> suppresses this check. This is useful with parameters which are known for reading back in a wrong way, frequent messages in the log can be avoided this way.</p>
<b>SAVE</b>	<p>The <code>SAVE</code> modifier tells the driver to save this variable on disk and to restore it's value when the program starts. Usually all device settings are stored in the device itself, the MNC system does not change anything at a device when it starts. In some cases, e.g. with <code>SETUP</code> variables or if the driver implements a receive level threshold, it is useful to store variable values in the MNC system rather in the device.</p>
<b>LOGPARAMETERCHANGESON/OFF</b>	<p>Defines if this variables shall log changes which are commanded to it. Parameter changes are logged if this is set ON and the <code>logParameterChanges</code> switch in the device setup page is activated as well. <code>LOGPARAMETERCHANGES</code> is <code>ON</code> by default for all variables, hence you only need to state <code>LOGPARAMETERCHANGES OFF</code> for variables you want explicitly to be excluded from logging.</p>

## LOGDETECTEDCHANGESON/OFF

Defines if this variables shall log changes which are detected when reading back the values. Detected changes are logged if this is set ON and the `logDetectedChanges` switch in the device setup page is activated as well.

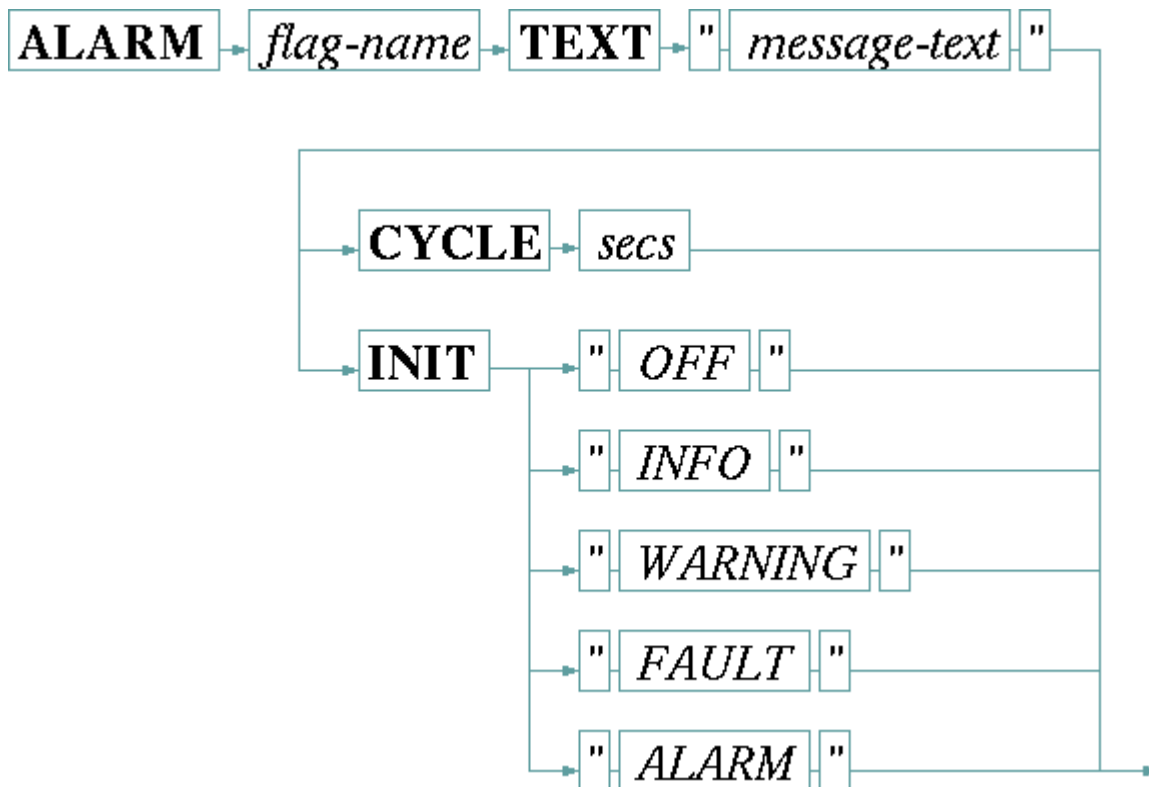
`LOGDETECTEDCHANGES` is `ON` by default for all variables, hence you only need to state `LOGDETECTEDCHANGES OFF` for variables you want explicitly to be excluded from logging. Please note, the this modifier has no effect on read-only variables, they are not logged regardless of the `logDetected` setting.

### Example

```
VAR audio.1.dotprogram    CYCLE 0  INTEGER 0 0 ""
VAR audio.1.dotrouting    CYCLE 0  CHOICE "NRM,MON,LFT,RGT"
VAR audio.1.dotoutput     CYCLE 0  CHOICE "ANALOG,AES/EBU,SPDIF,AC3"
VAR audio.1.dotlevel      CYCLE 0  INTEGER 6 18 "dB"
VAR audio.1.dotlanguage   CYCLE 0  TEXT
VAR audio.1.dottest       CYCLE 0  CHOICE "NRM,TEST-1,TEST-2,TEST-3,TEST-4,TEST-5"
VAR audio.1.dotinfo       CYCLE 4  TEXT READONLY
```

### 1.2.2.2 The ALARM statement

Alarm flags are a special variant of variables. They always are of the type `BOOL` and they are `READONLY`. For each `ALARM` variable you define a text message which appears in the *Faults* page of the device's user interface screen and as a message in the event log if the value of the flag changes. The device driver summarizes all alarm flags with each cycle and generates a summary fault information for the device. Alarm flags are defined with the `ALARM` statement:



An alarm flag definition consists of the **ALARM** keyword, the flag's name and an optional **CYCLE** definition which work analogous to the **CYCLE** modifier in a **VAR** statement.

The name of an alarm flag must be of the form **faults.XX** where **XX** is a two digit number in the range from 01 .. 98. The **faults.99** is reserved for a communication fault.

The alarm flag definition may include an **INIT** clause which initializes the fault priority value of the flag. **INIT** must be followed by one of **OFF** , **INFO** , **WARNING** , **FAULT** or **ALARM** .

### Example

```

ALARM faults.01  TEXT "Remote access" INIT "WARNING"
ALARM faults.02  TEXT "Synthesizer"
ALARM faults.03  TEXT "LO-A lock"
ALARM faults.04  TEXT "LO-B lock"
ALARM faults.05  TEXT "Power supply"
ALARM faults.06  TEXT "IF-LO level"
ALARM faults.07  TEXT "RF-LO level"
  
```

### 1.2.2.3 Info variables

Each device driver defines a number of so called info variables which tell the operator, but also the client software about the device driver. All info variables

- are named starting with **info.**
- are defined **READONLY** The user interface contains a special screen which display the

contents of all info variables the driver defines.

Info variables are used to display information like device types, serial numbers etc. Beside this, there are three info variables each device driver **must** define in any case. They are:

<b>info.type</b>	This variable tells the user interface name of the device driver. It is used to select the subdirectory for device presets and to find the help screen for this device driver. <code>info.type</code> must be initialized to the name of the device driver which in fact is the file name with the trailing <code>.device</code> cut off.
<b>info.port</b>	This variable is set by the device driver to the name of the serial port which is used to access the device. This is for convenience, offering the operator this information without opening the device setup.
<b>info.frame</b>	The standard user interface uses the value assigned to this variable to select the 'device oriented' screen set for this device. <code>info.frame</code> must be initialized to name of the device frame definition file to be used with this device.

### Example

```
VAR info.type      CYCLE 0 TEXT READONLY INIT "Tandberg-Alteia"
VAR info.port      CYCLE 0 TEXT READONLY
VAR info.frame     CYCLE 0 TEXT READONLY INIT "IRD-Alteia"
VAR info.model     CYCLE 0 TEXT READONLY
VAR info.serial    CYCLE 0 TEXT READONLY
```

The example above - taken from the Tandberg-Alteia device driver - identifies the driver as `Tandberg-Alteia` and tells the user interface to use the device oriented frame set called `IRD-Alteia` for this device. The info variables `info.model` and `info.serial` are set by the driver later, when it read the model description and serial number from the device.

#### 1.2.2.4 Setup variables

Another group of variables which are treated specially by the driver are setup variables. They are used to configure the device driver or to set parameters which shall not appear at the standard user interface.

Setup parameters are shown in an own screen which is accessible only to operators of a privilege level of 150 and above. Setup variables must be marked with the `SETUP` modifier, the name of setup variables must start with `.config`.

### Example

```
VAR config.lbandInputs  CYCLE 0 CHOICE "4,2" SETUP SAVE
VAR config.lnbPower     CYCLE 0 CHOICE "OFF,ON.,BST" SETUP
VAR config.loFreq       CYCLE 0 FLOAT 0 0 1 "MHz" SETUP
VAR config.berThreshold CYCLE 0 TEXT SETUP
VAR config.sigThreshold CYCLE 0 INTEGER 0 255 "" SETUP
VAR config.errFrame     CYCLE 0 CHOICE "FREEZE,BLACK" SETUP
```

The example above - taken from the `Tandberg-Alteia` device driver - defines a setup variable `config.lbandInputs` which configures the driver for 2- or 4-input models of the IRD. This variable also is marked with the `SAVE` modifier to make the setting permanent. The following variables in the example are configuration settings which are placed in the setup area to keep them out from the common user interface.

### 1.2.2.5 Variables for internal use

Variables may be used to store intermediate values in the driver. For instance, a bit field status value may be read into a variable and then interpreted using the `BITSET` command.

Variables for internal use must be declared `READONLY` to work properly. Even though not required, it is recommended to give these variables a descriptive name which tells about the internal usage of this variable. The following example is taken from the SSE KStar device driver:

```
VAR internal.state HEX 0 0 " " READONLY  CYCLE 2

...

PROC GET WATCH internal.state
  PRINT "AL"
  INPUT "=" internal.state
  BITSET faults.01 = internal.state 9
  BITSET faults.02 = internal.state 15
  BITSET faults.03 = internal.state 11

...
```

### 1.2.3 Using conversion tables

`CHOICE` parameter often are implemented by the device's remote control protocol as numeric constants (0 for BPSK, 1 for QPSK) or they are abbreviated in a way which is hard to remember. The user interface of the software shall in contrast display self-explaining names for such settings.

The driver language lets you define translation tables which do such conversions while the driver performs a `PRINT`, `INPUT`, `READ` or `WRITE` statement. As the driver knows, in which direction a table translation must be done, you can use the same table to translate values when they are written to and when they are read from the device.

The [TABLE](#) statement described in following chapter is used to define translation tables.

### 1.2.3.1 The TABLE statement

The driver language lets you define translation tables which do such conversions while the driver performs a `PRINT`, `INPUT`, `READ` or `WRITE` statement. Tables are defined with the `TABLE` statement:

```
TABLE → table-name → " table-definition "
```

The `TABLE` keyword is followed by a name for this table and the table definition in double quotes. The table definition contains assignments `A=B`, separated by commas. Strings containing commas or equal characters cannot be translated through a table.

```
TABLE tModulation "BPSK=BPS,QPSK=QPS,8PSK=8PS"
```

In the table definition, the `A` value is the string visible at the user interface of the software, the `B` value is used when communication with the device. The driver translates in a `PRINT` or `WRITE` statement `BPSK -> BPS`, `QPSK -> QPS` or `8PSK -> 8PS` if the table `tModulation` shown in the example above is referenced. With an `INPUT` or `READ` statement, the table automatically translates the values the other way round.

For some applications a table definition may become quite large. The driver syntax permits to split up the table definition in multiple string/lines. Each string/line must be enclosed in double quotes and must contain at least one translation pair. Commas at the end of a line are omitted. Below an example for such a multi line table definition is shown.

```
TABLE tVideoTest "NRM=00,625.1=01,625.2=02,625.3=03,625.4=04,625.5=05"
                  "625.6=06,625.7=07,625.8=08,625.9=09,625.10=10,625.11=11"
                  "625.12=12,625.13=13,625.14=14,625.15=15,625.16=16"
                  "525.1=17,525.2=18,525.3=19,525.4=20,525.5=21,525.6=22"
                  "525.7=23,525.8=24,525.9=25,525.10=26,525.11=27,525.12=28"
                  "525.13=29,525.14=30,525.15=31,525.16=32"
```

#### Remarks

- If a table translation is invoked with a value which is not contained in the table, the translation returns the first value in the table.
- To define a table it may be useful to copy the definition of the `CHOICE` variable which shall be translated at the place of the table statement. this copy then can be extended to the table definition.
- Don't add whitespace characters in the table definition to make this better readable. The driver would treat these characters as part of the strings to translate!

### 1.2.4 Adding data exchange procedures

While the device driver variables discussed in the chapters above specify the driver's interface



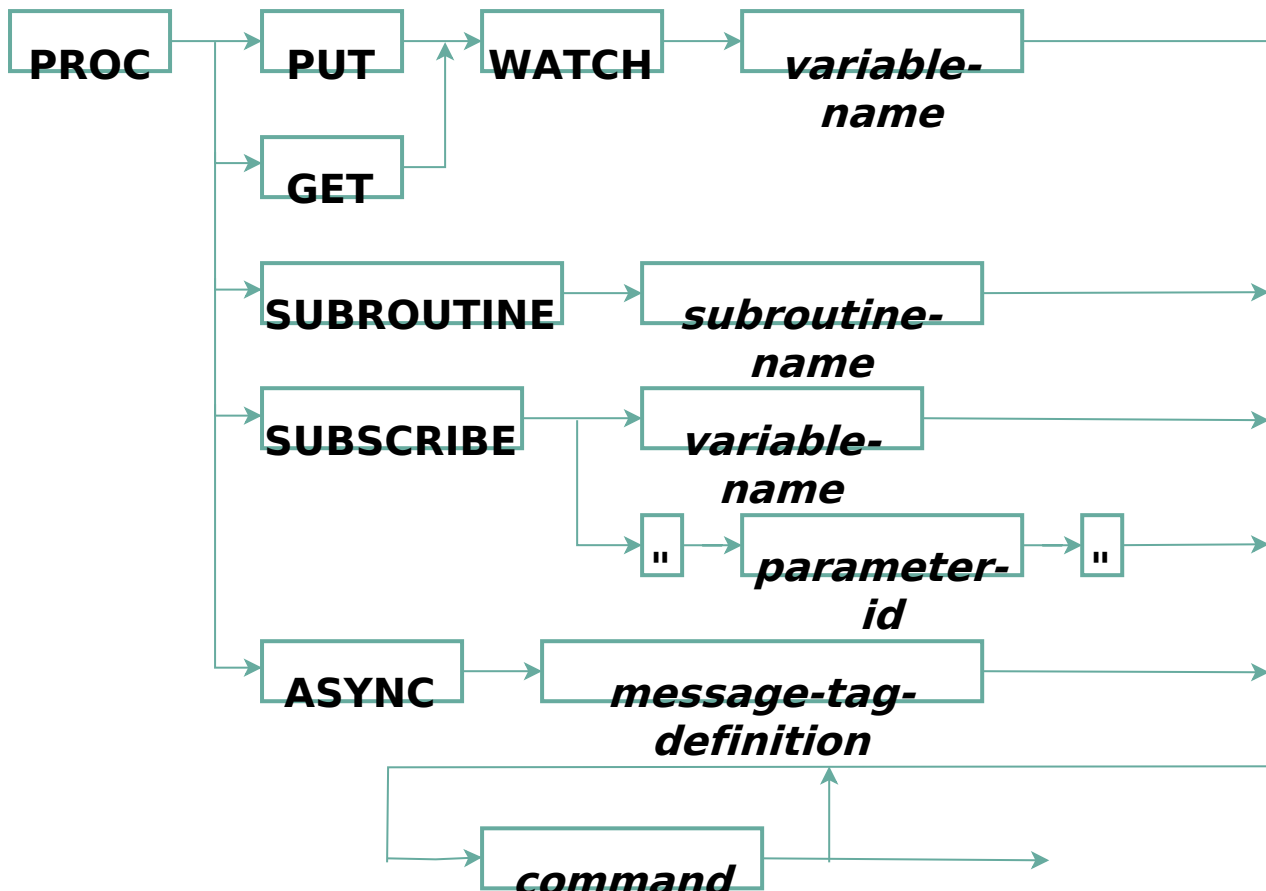
to the other parts of the software, does the device driver procedure implement the interface to the device itself. Based on the communication protocol selected in the equipment setup (the protocol handles the low level coding of the data exchanged with the device), procedures handle the chats done between the MNC and the device to perform parameter settings or to gain some information from the device.

The description of the [PROC](#) statement in the following chapter together with the examples supplied with this manual explain the usage of procedures. Generally the following rules apply to device driver procedures:

- Procedures are executed by the driver in the same order as they are defined in the driver specification.
- The driver knows GET-type procedures which read some information from the device into one or more variables and PUT-type procedures which send settings to the device.
- A procedure may handle one or more parameters.
- The driver skips any procedures where actually nothing is to do. For example a PUT-type procedure is skipped unless at least one of the values handled by it has been changed by the operator.
- A variable which has it's data source in the device must be handled at exactly one GET-type procedure.
- A variable being a device setting must be handled at exactly one PUT-type procedure.

#### **1.2.4.1 The PROC statement**

Driver procedures are defined with the PROC statement. The PROC keyword is followed by one of `PUT` , `GET` , `SUBSCRIBE` , `SUBROUTINE` or `ASYNC` . Chapter [Using subroutines](#) tells more about this type of procedures. `ASYNC` procedures handle messages sent by the device in an unsolicited way. Chapter [ASYNC procedures](#). Procedures do not have an explicit *end* keyword, a procedure automatically ends where another definition (procedure, variable or table) starts.



`PUT` and `GET` procedures must have a `WATCH` keyword followed by one or more names of variables this procedure shall be bound to.

Generally, any variable which is referenced in a procedure must be defined before using the `VAR` statement. Although the driver language allows to define variables between procedures (a `VAR` statement closes an open procedure definition), the drivers supplied by SatService GmbH first define all variables and then all procedures for this reason.

## Examples

There are two complete device drivers listed at the end of this section, showing several flavors of procedure definitions:

- [NDSatCom-KuBand-Upconverter device driver](#)
- [Tandberg-Alteia device driver](#)

## Variable subscriptions

The `PROC SUBSCRIBE` variant of procedure definitions permits to react on changes of parameters controlled by other devices drivers. Such a procedure is called every time the referenced parameter changes. The parameter to listen to may be defined directly with its full parameter name in double quotes (e.g. `DEVICE.some.name`) or by the name of a variable in this device driver which contains the name of the parameter to listen to.

SUBSCRIBE type procedures are executed in the thread context where the parameter it's listening to is changed, not in the thread context of the local device driver. For this reason, SUBSCRIBE procedures never should contain IO-related statements like PRINT , INPUT , READ , oder WRITE . Communicating to the device from different program threads may break the communication protocol. That is much similar like when two people are talking at the same time, probably you won't understand any of them.

#### 1.2.4.2 ASYNC procedures

Device drivers may define ASYNC procedures which handle messages sent by a device without a foregone request sent by the MNC. While the *sat-nms* device driver architecture is targeted to devices which only reply if requested to do this, ASYNC procedures provide a ways to handle unsolicited messages.

##### Basic concept

The concept behind ASYN procedures is a communication protocol which listens to the device all the time, reads all messages coming from the device. Each message is forwarded to the ASYNC procedures defined to check which procedure is in charge for this particular message.

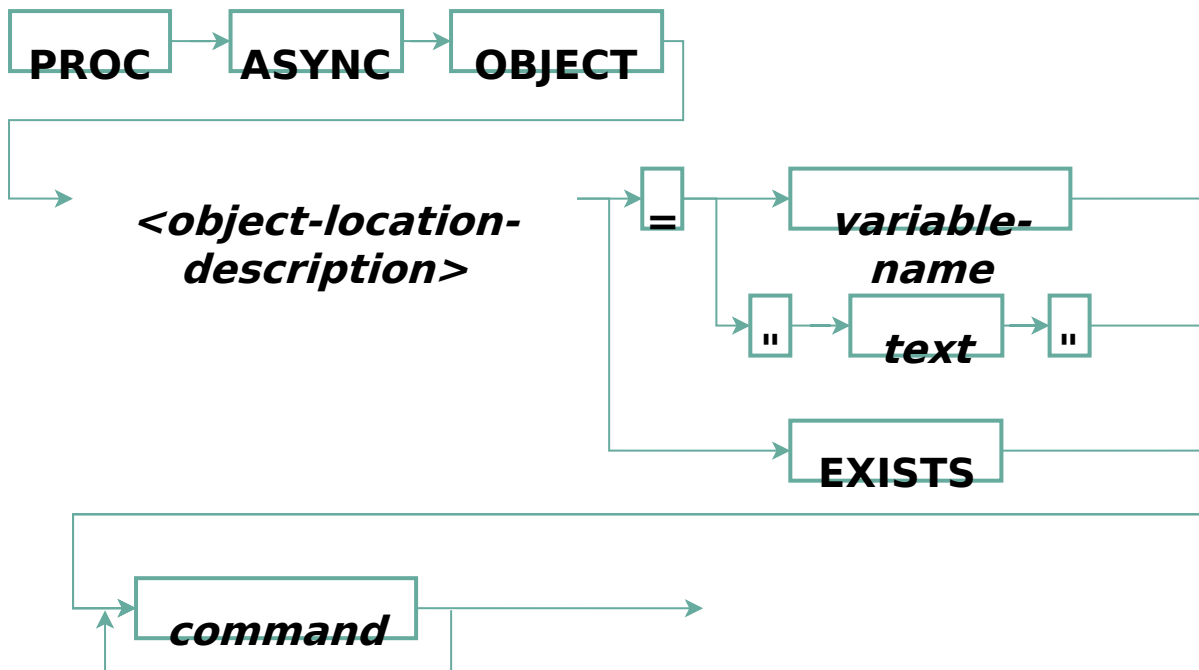
The check ist done by comparing so called message tags to a reference value defined for the procedure. For JSON based protocols the message tag is one value in the JSON document.

There are a number of constraints you have to consider if a device sends messages without being requested to, specially if the device responds to requests *and* sends unsolicited messages as well:

1. Reading data from a device asynchronously requires a *sat-nms* communication protocol definition that supports this. The protocol must listen to the device all the time and forward any incoming message to the device driver. If the protocol does not support this, no ASYNC procedures will be executed in a driver, even if they are defined.
2. You can no longer assume, that reading a device's reply after sending a request contains the expected data. You have to design the driver in a way, that all commands and status requests are sent in PUT procedures without reading the device's reply. All replies and unsolicited messages have to be read in ASYNC procedures.
3. There are devices which provide separate communication channels for replies to requests and for asynchronous messages. With such a protocol the above restriction does not apply.
4. Commands and requests *never* must be sent from within an ASYNC procedure. As the ASYNC procedure runs asynchronously to the device driver thread, this may collide with commands sent from a PUT procedure.

##### PROC ASYNC for JSON coded protocols

For JSON based protocols the definition of an asynchronous procedure starts with the keywords PROC ASYNC OBJECT , followed by an object path definition as used with the REST SET and REST PARSE statements. You find an extensive description about the JSON object model and object paths in chapter [2.2.6 REST I/O functions](#)

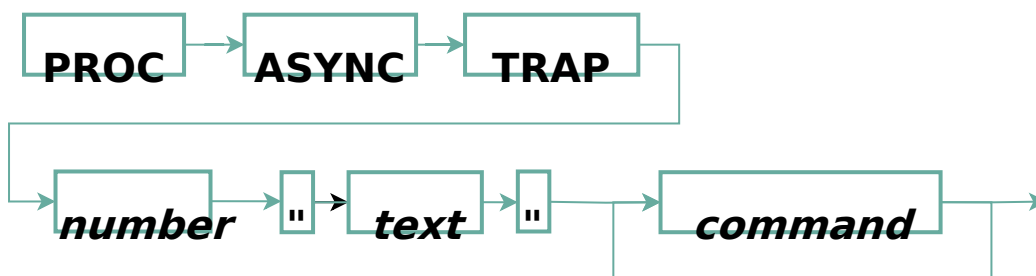


The object path definition is terminated either by the keyword `EXISTS` or a `=` character and the reference value this JSON object gets compared to. The reference value may be a quoted string or the content of a driver variable. The procedure body gets executed if the JSON object defined by the object path contains a value equal to the reference value following the `=` character.

If the keyword `EXISTS` is used, the procedure gets executed if an object at the given location exists in the message, the value assigned to it doesn't care.

### PROC ASYNC for SNMP protocols

With SNMP, `PROC ASYNC TRAP` may be used to process traps issued by a device. The syntax for this kind of procedure is as follows:



The keywords `PROC ASYNC TRAP` are followed by the SNMP trap number and a pattern text. The SNMP trap calls this procedure if it has been issued from the IP address set for this device, the SNMP trap number matches the number stated with this procedure and the text submitted by the trap receiver contains the pattern text.

The trap receiver in the M&C software processes every trap it receives and translates it to a multiline string which is passed to the procedure(s) which are in charge for this trap. The first

line of this string contains the trap type, the IP address from which it was issued, the trap number and the specific number. If there are SNMP varbinds with this trap, for each varbind a line follows with the OID and the value if this varbind.

Example:

```
V1TRAP from 127.0.0.1 trapno=6 specific=99
1.3.6.1.4.1.53491.550.2.3.19.0=-76.33
1.3.6.1.4.1.53491.550.2.3.24.0=2
```

The varbinds may be interpreted with `INPUT` statements in the procedure body:

```
PROC ASYNC TRAP 6 "53491.550.2.3.19.0"
  INPUT "53491.550.2.3.19.0=" TRM 10 inputLevel
  INPUT "53491.550.2.3.24.0=" TRM 10 XLT tLock inputLock
```

The procedure above gets called for a trap (6) containing a varbind ending on "53491.550.2.3.19.0". The value of this varbind gets assigned to the driver variable *inputLevel*. The varbind ending on "53491.550.2.3.24.0" is assigned to 'inputLock' after translating it from the SNMP number encoding to the value expected in the driver. Using OID fragments starting with the device's enterprise ID is a reliable way to identify varbind OIDs in the trap text.

## Remarks

- The trap number in the procedure header may be set to \* to make the procedure accept any trap number. For SNMPv3 traps / informs the trap number must be \* as SNMPv3 traps do not contain trap numbers.
- The pattern text in the procedure header may be set to "\*" to accept any trap without checking if it contains the pattern text.
- `PROC ASYNC TRAP` procedures require the M&C server's trap receiver to setup properly. The file 'traprec.json' configures the properties of the trap receiver, the M&C user manual gives a comprehensive description of this configuration file.

## 1.2.5 Basic I/O functions

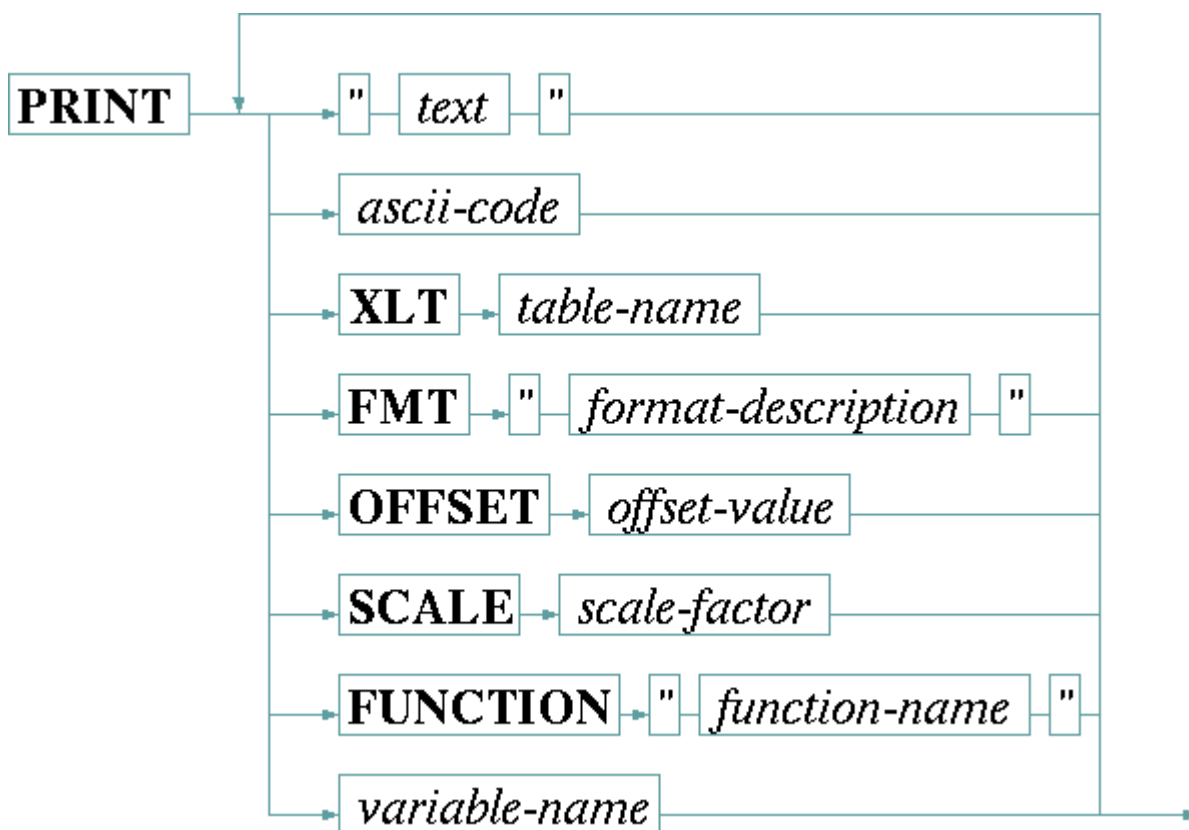
The primary function of a driver procedure is to acquire some information from the device (`GET` procedure) or to set one or more parameters from the data commanded somewhere else in the software (`PUT` procedure). The **sat-nms** device driver definition language provides four highly flexible statements for this purpose. They are:

<a href="#">PRINT</a>	Composes/formats s (readable) string from multiple elements which may be text fragments, numbers (formatted in various ways), tokens from a <code>CHOICE</code> variable and more. The composed string is packed in a protocol frame and sent to the device. <code>PRINT</code> is used with devices which implement a text based protocol.
-----------------------	---

<a href="#">INPUT</a>	The <b>INPUT</b> statement complements the <b>PRINT</b> statement. It reads a message from the device, strips off the protocol frame parses the received data according to the rules specified with the <b>INPUT</b> statement.
<a href="#">WRITE</a>	The <b>WRITE</b> statement is the equivalent to <b>PRINT</b> for devices providing a binary communication protocol. <b>WRITE</b> sets up a binary data structure from constant and variable values, packs this structure in a protocol frame and sends it to the device.
<a href="#">READ</a>	Reading data from a device providing a binary communication protocol is done with the <b>READ</b> statement. As the counterpart of the <b>WRITE</b> statement it reads a message from the device, strips off the protocol frame and provides means to interpret the received data as a binary structure variables.

### 1.2.5.1 The PRINT statement

The **PRINT** statement is used to send commands to a device which uses a text based protocol. It composes a command string from the elements following the the **PRINT** keyword, packs this string in the protocol frame defined by the communication protocol and sends this message to the device. The syntax of the **PRINT** statement is:



The table below describes the elements which may follow the **PRINT** keyword.

<b>"text"</b>	Plain text, enclosed in double quotes, is added to the output buffer as it is.
---------------	--

<b>ascii-code</b>	A decimal number is interpreted as the ASCII code of a single character to output. You may use this to send special characters like carriage-return or line-feed to the device.
<b>XLT table</b>	The <code>XLT</code> keyword, followed by the name of an already defined table, tells the driver to translate the next variable value which shall be printed through this translation table. Chapter <a href="#">Using conversion tables</a> gives more information about tables in general.
<b>FMT "..."</b>	The <code>FMT</code> keyword, followed by a format description in double quotes, tells the device driver to format the next variable following the format description given. <code>FMT</code> is used to format numeric variables into the representation the device expects in it's remote control command. <code>INTEGER</code> variables may be printed using a floating point format, and <code>FLOAT</code> variables may be printed with a <code>d</code> or <code>x</code> format likewise. With <code>FLOAT</code> variables you should format in any case as internal precision/rounding problems may cause unpredictable results when floating point values are printed unformatted.

A format description consists of the following elements:

<b>format characters</b>	<b>description</b>
<b>d b x X f</b>	The first letter of the format description defines the general number representation: <code>d</code> (decimal), <code>b</code> (binary), <code>x</code> (hex, lower case), <code>X</code> (hex, upper case) or <code>f</code> (floating point)
<b>+</b>	If the <code>+</code> option is given, the number is preceded by a +/- character even if it is positive. Together with the <code>0</code> option below, the sign appears as an additional character at the first column, hence the field width is increased by one on this case.
<b>0</b>	If the <code>0</code> option is given, the field is padded up with zeroes instead of spaces.
<b>1 .. 99</b>	Here follows a one or two digit field width. The field width is the total number of characters the formatted number occupies including the padding characters. If there are more digits needed to show a number correctly, the field with is enlarged automatically. Specifying a field width <code>1</code> disables the right orientation in a fixed width completely. This specially is useful for floating point numbers where only the number of fraction digits shall be fixed, not the complete field width.
<b>.</b>	For floating point formats the dot separates the precision from the field width.

format characters	description
<b>0 .. 9</b>	The dot is followed by a one digit specification of the number of fraction digits which shall be printed. The dot and fraction digits specification is valid only with the floating point format.

The table below describes the more elements which may follow the **PRINT** keyword.

<b>OFFSET o</b>	The <b>OFFSET</b> keyword, followed by the (floating point) offset value, tells the driver to add the offset value to the next variable before it is printed.
<b>SCALE s</b>	The <b>SCALE</b> keyword, followed by the (floating point) scale factor, tells the driver to multiply the next variable with the scale factor before it is printed
<b>FUNCTION "name"</b>	The <b>FUNCTION</b> keyword, followed by the name of the function as a quoted string, tells the driver to apply this function to the next variable before it is printed. The function name is extended by appending <b>.txt</b> to achieve the file name to read for the function definition. Chapter <a href="#">Function tables in I/O functions</a> gives an extensive description about functions and how they are used.
<b>variable-name</b>	A variable name tells the driver to print the value stored in this variable. Usually the <i>commanded</i> value is used rather than the value which has been read back from the device. If, however, there has been never a value commanded, or if this variable is a read only variable, the value recently read from the device is used.

Before the variable is printed to the output buffer, any **XLT** , **FMT** , **OFFSET** or **SCALE** operations which have been specified are applied. This happens in a fixed order:

1. If a **SCALE** operation has been specified, this is done first.
2. The **SCALE** operation is followed by an **OFFSET** addition.
3. The so scaled value is processed through a **FUNCTION** if there is one.
4. The processed value is formatted according to a **FMT** specification if one is given.
5. Finally, the value gets translated through a translation table, if an **XLT** operation has been specified.

This scheme applies to numeric ( **INTEGER** , **HEX** , **FLOAT** ) and to text type variables as well. As soon as a **SCALE** , **OFFSET** , **FUNCTION** or **FMT** keyword is present in front of a variable, this gets converted to a floating point number. Strings which cannot be converted give a zero value.

### Example

```
PRINT "TUN;FRQ=" SCALE 8.0 FMT "d06" frequency
```

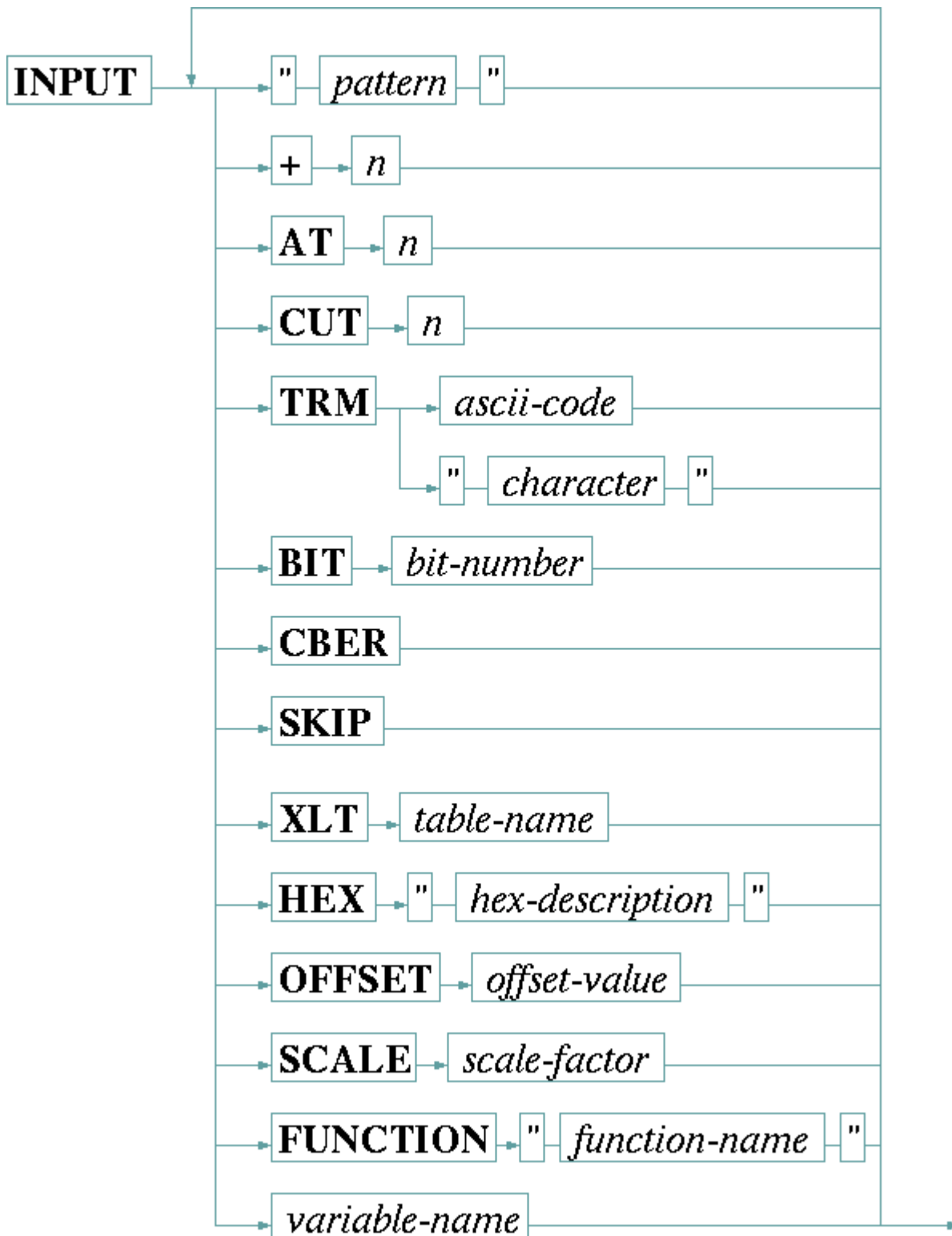
The example above sends a command of the format **TUN;FRQ=#####** to the device. The



`Float` variable `frequency` is multiplied by `8.0` and the output as an integer number, 6 digits wide with leading zeroes.

#### 1.2.5.2 The **INPUT** statement

The `INPUT` statement is used to read and interpret the reply of a device which uses a text based communication protocol. It reads a protocol frame from the device and parses the received message according to the rules defined by the description following the `INPUT` keyword. The syntax of the `INPUT` statement is:



While parsing the device's reply, the driver deals with three information buffers.

1. *original reply*
2. *pad* buffer
3. *value* buffer

To understand the way how the parsing operations work which may follow the `INPUT` keyword, it is necessary to learn how the driver uses these buffers.

- The first buffer contains the original reply string as it is received from the device (with the protocol frame stripped off).
- The driver initially copies this string into a second buffer called *pad*. Search operations (text in double quotes) and the "+" operation modify the *pad*. The AT operation is the only way to restore the *pad* from the original data.
- Every time, the *pad* is modified, the driver copies it's contents into the third buffer called *value*. *Value* is the string which will be assigned to a variable if it appears in the list of arguments to the `INPUT` statement. Operations in the `INPUT` statement like `CUT`, `TRM`, or `XLT` work on the *value* buffer but leave the *pad* unchanged. This is important to understand if you have to parse the value of more than one variable in a single `INPUT` statement.

<b>" pattern "</b>	Searches the text/pattern in the current <i>pad</i> buffer. If the pattern is found, all text including the first occurrence of the pattern is removed from the beginning of the <i>pad</i> buffer. The pattern text may contain escape sequences like <code>\n</code> or <code>\t</code> to search for line feeds or tabulator characters.
<b>+ n</b>	Removed <code>n</code> characters from the beginning of the <i>pad</i> buffer.
<b>AT n</b>	Restores the <i>pad</i> buffer with a substring of the original data, starting at column <code>n</code> .
<b>CUT n</b>	Cuts the <i>value</i> buffer to a length of <code>n</code> characters.
<b>TRM "c"</b>	Removes the first occurrence of the termination character and all following text from the <i>value</i> buffer. The termination character may be specified as a decimal number (e.g. 10 is the ASCII code for line feed) or as a single character enclosed in double quotes.
<b>BIT n</b>	Interprets the <i>value</i> buffer as a decimal number and isolates the bit number <code>n</code> ( <code>n = 0</code> means the least significant bit). The value buffer is replaced by <code>0</code> or <code>1</code> depending on the bit value.
<b>CBER</b>	Interprets the <i>value</i> buffer as a bit error rate value as it is returned by Comstream modems. A string <code>mn</code> is converted into the more common notation <code>mE-n</code> .
<b>SKIP</b>	Removes any whitespace (space characters, line feeds, carriage returns or tabs) from the beginning of the <i>value</i> buffer.
<b>XLT table</b>	The <code>XLT</code> keyword, followed by the name of an already defined table, tells the driver to translate the <i>value</i> buffer through this translation table from right to left. Chapter <a href="#">Using conversion tables</a> gives more information about tables in general.

<b>OFFSET o</b>	The <code>OFFSET</code> keyword, followed by the (floating point) offset value, interprets the <i>value</i> buffer as a floating point number and replaces the buffer with the sum of the value and the offset.
<b>SCALE s</b>	The <code>SCALE</code> keyword, followed by the (floating point) scale factor, interprets the <i>value</i> buffer as a floating point number and replaces the buffer with the product of the value and the scale factor.
<b>FUNCTION "name"</b>	The <code>FUNCTION</code> keyword, followed by the name of the function as a quoted string, tells the driver to apply this function to the next variable before it is printed. The function name is extended by appending <code>.txt</code> to achieve the file name to read for the function definition. Chapter <a href="#">Function tables in I/O functions</a> gives an extensive description about functions and how they are used.
<b>variable-name</b>	A variable name tells the driver to assign the contents of the <i>value</i> buffer to this variable. The driver performs format conversions and range checks according to the type and limit specifications made in the VAR definition of this variable.
<b>HEX ".."</b>	Interprets the contents of the <i>value</i> buffer as a hex number and replaces it by its decimal equivalent.

The format definition string which must follow the HEX keyword may contain the following characters:

<b>L</b>	The hex number is in <i>little endian</i> notation. This means the least significant byte is written at first. If no <code>L</code> character is in the format string, <i>big endian</i> notation is assumed.
<b>S</b>	The hex number is signed. The number of digits is used to calculate the sign extension: <code>80</code> for example is calculated as <code>-1</code> as the two digit number is assumed to be a byte value. <code>0080</code> in contrast is computed to <code>128</code> as a 16 bit number of this value is positive.

## Remarks

If a value is not assigned to a variable as you expect, you should check the following issues:

- Numeric values are converted from string to numeric representation in a very lenient way. For instance, any leading non-numeric characters are ignored. If a numeric variable is not assigned as expected, in most cases there is a range violation. The driver does not accept values outside the range defined for this variable.
- `CHOICE` variables and tables (referenced with the XLT operation) expect an input value which *exactly* matches one of the valid choices. See chapter [Using conversion tables](#) to read how you can make tables somewhat fault tolerant.

## Example

```

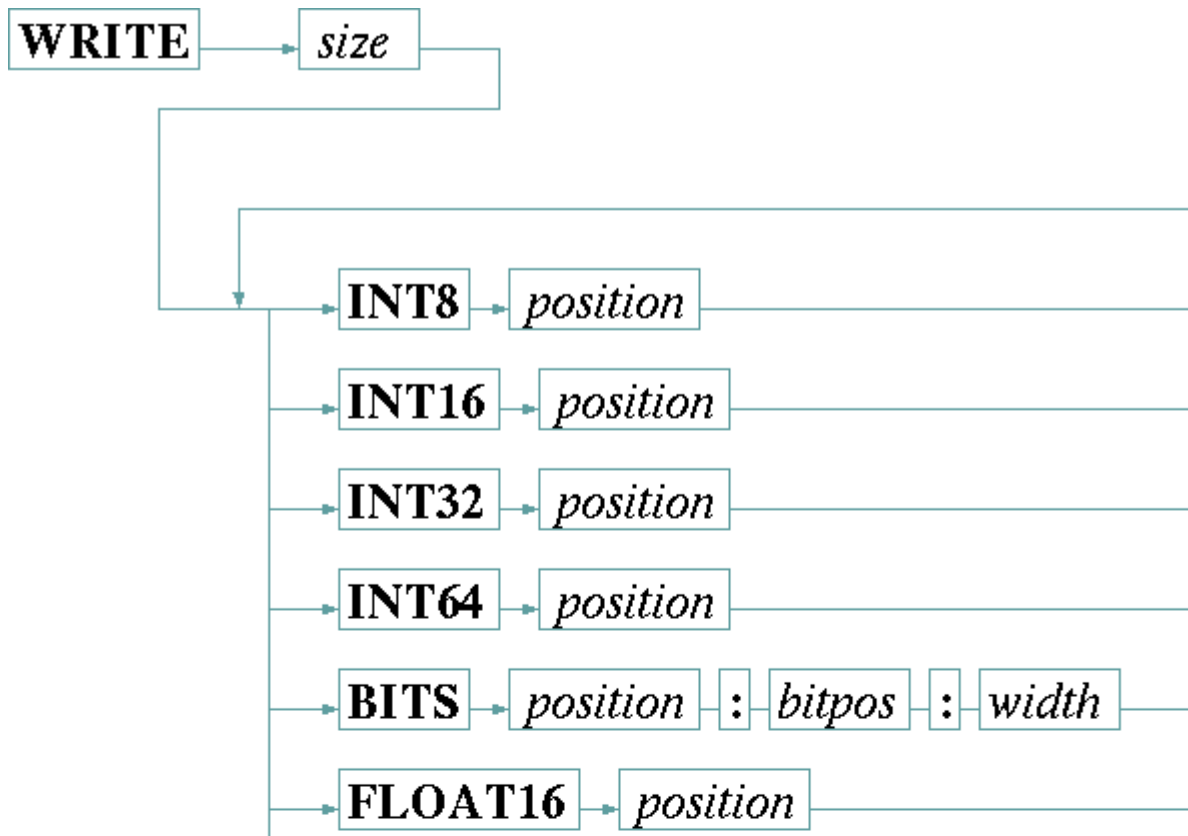
INPUT AT 2 SCALE 0.001      tx.frequency
  AT 11 SCALE -0.1 OFFSET 30.0 tx.gain
  AT 15 CUT 1 XLT T03      faults.01
  AT 17 CUT 1 XLT T02      info.if
  AT 19 CUT 1 XLT T01      tx.on
  AT 21 CUT 1              faults.02
  AT 22 CUT 1              faults.03
  AT 23 CUT 1              faults.04
  AT 24 CUT 1              faults.05
  AT 25 CUT 1              faults.06
  AT 26 CUT 1              faults.07

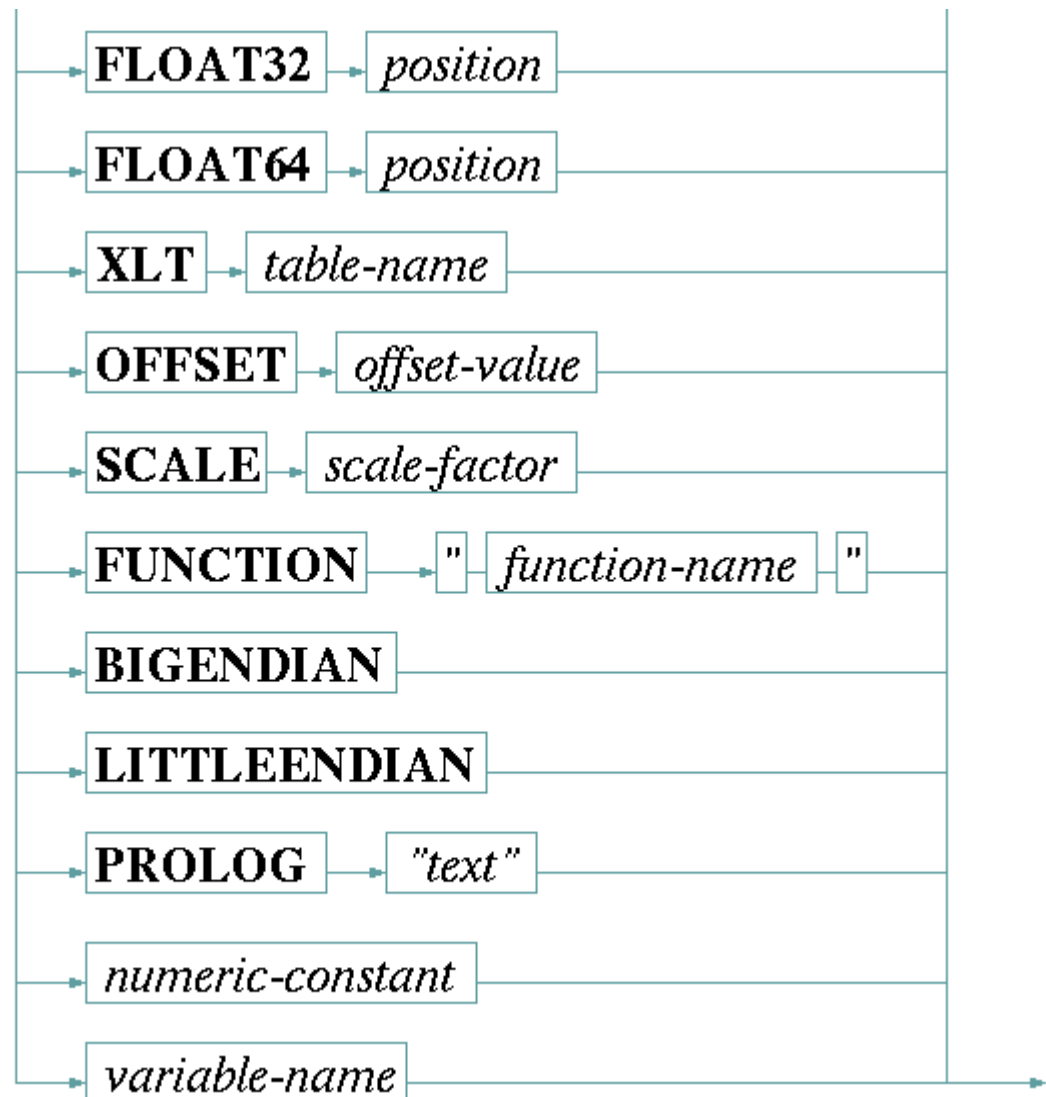
```

The example above - taken from the [NDSatCom-KuBand-Upconverter](#) device driver - parses all settings and faults flags from a single status string the upconverter returns. Note that the numeric variables `tx.frequency` and `tx.gain` are isolated from the string without specifying a length or a termination character. The number recognition routine automatically stops where it finds a non-numeric character after the number to read. The other values however all are cut to one character length before they are interpreted.

### 1.2.5.3 The WRITE statement

The `WRITE` statement is used to send data to the device which uses a binary communication protocol. Such a device expects a command as a binary data structure rather than as a readable text. The syntax of the `WRITE` command is:





The **WRITE** statement creates a buffer of the specified number of bytes which initially is filled with zeroes. Variables or constant values are copied into this buffer according to the size and position specifications given.

The **WRITE** statement by default works in *little endian* mode. 16, 32 or 64 bit values are copied into the buffer with the least significant byte first. Specifying **BIGENDIAN** mode reverses this byte order.

<b>size</b>	The size (number of bytes) of the message buffer to create must be specified as the first argument following the <b>WRITE</b> keyword. The size value is expected as a decimal number.
<b>INT8 p</b>	The <b>INT8</b> keyword, followed by a decimal position value, tells the driver that the next constant or variable shall be copied to the buffer as a single byte value at the given position (position zero denotes the first byte in the buffer).

<b>INT16 p</b>	The <b>INT16</b> keyword, followed by a decimal position value, tells the driver that the next constant or variable shall be copied to the buffer as a two-byte value starting at the given position (position zero denotes the first byte in the buffer). The least significant byte of the number is placed first unless <b>BIGENDIAN</b> was specified.
<b>INT32 p</b>	The <b>INT32</b> keyword, followed by a decimal position value, tells the driver that the next constant or variable shall be copied to the buffer as a four-byte value starting at the given position (position zero denotes the first byte in the buffer). The least significant byte of the number is placed first unless <b>BIGENDIAN</b> was specified.
<b>INT64 p</b>	The <b>INT64</b> keyword, followed by a decimal position value, tells the driver that the next constant or variable shall be copied to the buffer as a eight-byte value starting at the given position (position zero denotes the first byte in the buffer). The least significant byte of the number is placed first unless <b>BIGENDIAN</b> was specified.
<b>BITS p : b : w</b>	The <b>BITS</b> keyword tells the driver to place the next value as a 1 to 7 bit wide bit field within a certain byte. In this case the position parameter consists of three values, separated by colons:
	<b>position</b> <b>p</b> addresses the byte within the buffer.
	<b>bitpos</b> <b>b</b> defines a number of bit positions the value shall be shifted left before it is pasted into the byte.
	<b>width</b> <b>w</b> the number of bits (starting with the least significant one) which shall be copied from the next constant or variable into the destination byte.
<b>FLOAT16 p</b>	The <b>FLOAT16</b> keyword, followed by a decimal position value, tells the driver that the next constant or variable shall be copied to the buffer as a half precision (16 bit) IEEE754 floating point value starting at the given position (position zero denotes the first byte in the buffer). The least significant byte of the number is placed first unless <b>BIGENDIAN</b> was specified.
<b>FLOAT32 p</b>	The <b>FLOAT32</b> keyword, followed by a decimal position value, tells the driver that the next constant or variable shall be copied to the buffer as a single precision (32 bit) IEEE754 floating point value starting at the given position (position zero denotes the first byte in the buffer). The least significant byte of the number is placed first unless <b>BIGENDIAN</b> was specified.

<b>FLOAT64 p</b>	The <code>Float64</code> keyword, followed by a decimal position value, tells the driver that the next constant or variable shall be copied to the buffer as a double precision (64-bit) IEEE754 floating point value starting at the given position (position zero denotes the first byte in the buffer). The least significant byte of the number is placed first unless BIGENDIAN was specified.
<b>XLT table</b>	The <code>XLT</code> keyword, followed by the name of an already defined table, tells the driver to translate the next variable value which shall be printed through this translation table. When output by a <code>WRITE</code> statement, any non-numeric variable must be translated through a table into a numeric value. Chapter <a href="#">Using conversion tables</a> gives more information about tables in general.
<b>OFFSET o</b>	The <code>Offset</code> keyword, followed by the (floating point) offset value, tells the driver to add the offset value to the next variable before it is pasted into the buffer.
<b>SCALE s</b>	The <code>Scale</code> keyword, followed by the (floating point) scale factor, tells the driver to multiply the next variable with the scale factor before it is pasted into the buffer.
<b>FUNCTION "name"</b>	The <code>Function</code> keyword, followed by the name of the function as a quoted string, tells the driver to apply this function to the next variable before it is printed. The function name is extended by appending <code>.txt</code> to achieve the file name to read for the function definition. Chapter <a href="#">Function tables in I/O functions</a> gives an extensive description about functions and how they are used.
<b>BIGENDIAN</b>	The <code>BIGENDIAN</code> keyword turns the byte order from <i>little endian</i> (LSB first) which is the default to <i>big endian</i> (MSB first). This applies to all values written after BIGENDIAN.
<b>LITTLEENDIAN</b>	The <code>LITTLEENDIAN</code> keyword turns the byte order back to <i>little endian</i> (LSB first). This applies to all values written after <code>LITTLEENDIAN</code> .
<b>PROLOG "text"</b>	The <code>PROLOG</code> keyword followed by a quoted string sets this string as a <i>prolog</i> to be sent before the binary data specified in the <code>WRITE</code> statement. Actually, this is only usable together with the HTTP11 protocol to generate a HTTP 1.1 POST request. Example: <code>PROLOG "POST some/url "</code> prepends this text before the binary data.
<b>numeric-constant</b>	Numeric (decimal) constants can be used like variable names. The constant value is pasted into the output buffer in this case. If a quoted string is given, this is treated as a hexadecimal constant and translated into a numeric value.



<b>variable-name</b>	A variable name tells the driver to paste the value stored in this variable into the output buffer according to the size and position specification given before. Usually the <i>commanded</i> value is used rather than the value which has been read back from the device. If, however, there has been never a value commanded, or if this variable is a read only variable, the value recently read from the device is used.
----------------------	---

Before the variable is printed to the output buffer, any `XLT` , `OFFSET` or `SCALE` operations which have been specified are applied. This happens in a fixed order:

1. If a `SCALE` operation has been specified, this is done first.
2. The `SCALE` operation is followed by an `OFFSET` addition.
3. The so scaled value is processed through a `FUNCTION` if there is one.
4. The processed value is formatted according to a `FMT` specification if one is given.
5. Finally, the value gets translated through a translation table, if an `XLT` operation has been specified.

### Remarks

- Each variable name or constant must be preceded by an `INT8` , `INT16` , `INT32` , `INT64` or `BITS` specification to be pasted properly into the output buffer.
- Only `INTEGER` , `HEX` or `FLOAT` type variables can be used directly with the `WRITE` statement. For other variable types an `XLT` table translation must be used to turn the variable into a numeric value.
- `FLOAT` variables are converted to 64-bit integer values before they are used. `OFFSET` or `SCALE` operations are done before this conversion.

### Example

```
TABLE T02 "QPSK=0,BPSK=1"

...

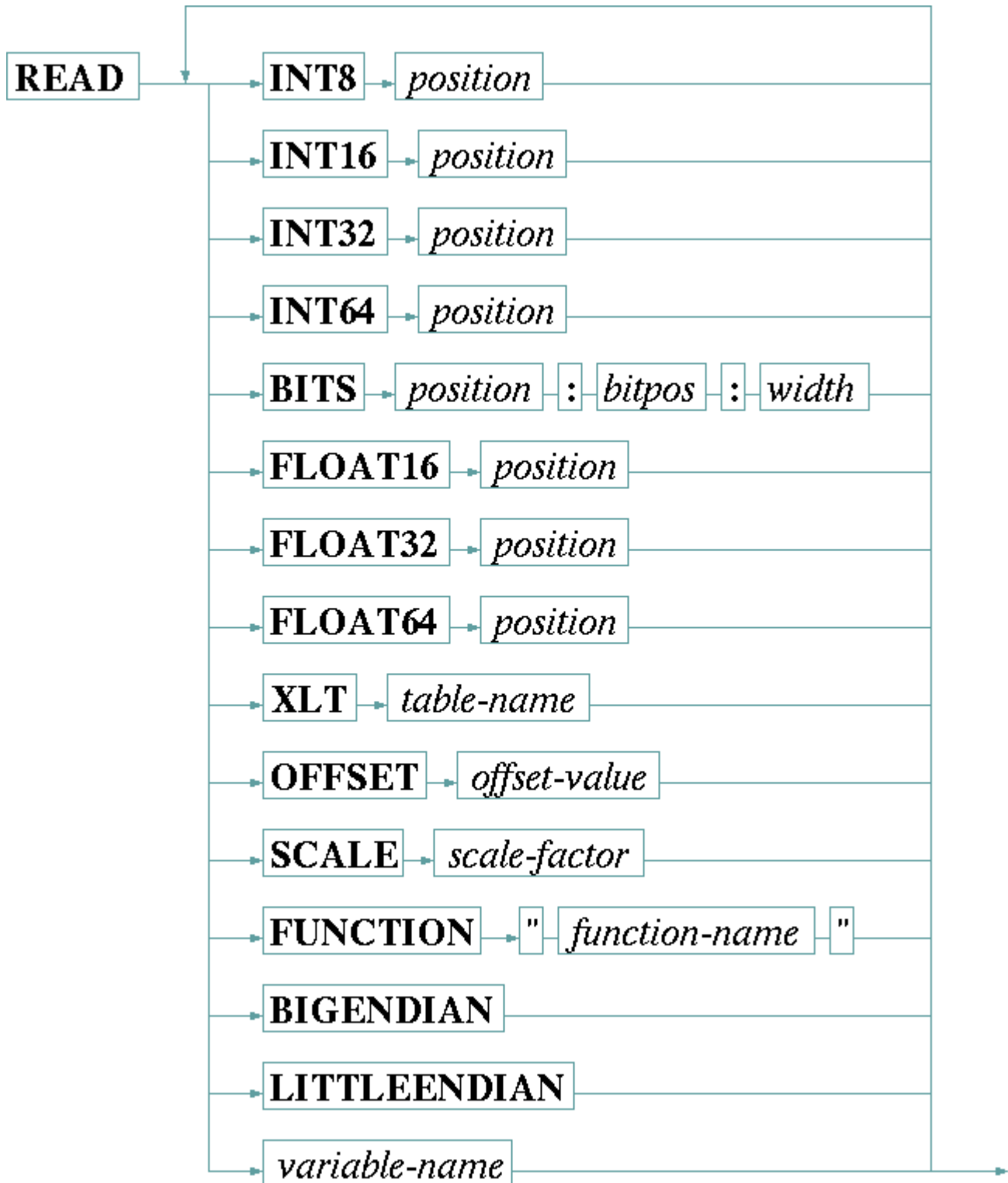
WRITE 3 BIGENDIAN
  INT16 0 9734
  INT8 2 XLT T02      tx.mod.type
```

The example above -- taken from the Radyne DVB3030 modulator driver -- explains how to write a `CHOICE` parameter to a device. Table `T02` is used to convert the QPSK/BPSK selection into the numeric values 0/1. The `WRITE` statement sends a message (three bytes long plus protocol frame overhead) in *big endian* byte order. The first two bytes are filled with the decimal constant `9734` (a command code). The third byte is set to `0` or `1` according to the contents of `tx.mod.type` .

#### 1.2.5.4 The READ statement

Receiving and interpreting of data returned from devices which use a binary communication

protocol is done with the **READ** statement. The **READ** statement gets a message from the device, strips off the protocol frame and places the result in an internal buffer. Subcommands/operations following the **READ** keyword are used to copy values from the buffer into one or more variables of the device driver.



The **READ** statement by default works in 'little endian' mode. 16, 32 or 64 bit values are read

from the buffer with the least significant byte first. Specifying **BIGENDIAN** mode reverses this byte order.

<b>INT8 p</b>	The <b>INT8</b> keyword, followed by a decimal position value, tells the driver that the next variable appearing in the argument list shall be filled from a single byte value at the given position (position zero denotes the first byte in the buffer).
<b>INT16 p</b>	The <b>INT16</b> keyword, followed by a decimal position value, tells the driver that the next variable appearing in the argument list shall be filled from a two-byte value starting at the given position (position zero denotes the first byte in the buffer). The least significant byte of the number is read first unless <b>BIGENDIAN</b> was specified.
<b>INT32 p</b>	The <b>INT32</b> keyword, followed by a decimal position value, tells the driver that the next variable appearing in the argument list shall be filled from a four-byte value starting at the given position (position zero denotes the first byte in the buffer). The least significant byte of the number is read first unless <b>BIGENDIAN</b> was specified.
<b>INT64 p</b>	The <b>INT64</b> keyword, followed by a decimal position value, tells the driver that the next variable appearing in the argument list shall be filled from a eight-byte value starting at the given position (position zero denotes the first byte in the buffer). The least significant byte of the number is read first unless <b>BIGENDIAN</b> was specified.
<b>BITS p : b : w</b>	The <b>BITS</b> keyword tells the driver to read a 1 to 7 bit wide bit field from a certain byte in the buffer. In this case the position parameter consists of three values, separated by colons:
	<b>position</b> <b>p</b> addresses the byte within the buffer,
	<b>bitpos</b> <b>b</b> defines the bit-number (0 to 7) of the least significant bit to read.
	<b>width</b> <b>w</b> the number of bits which shall be copied from the buffer into the destination variable.
<b>FLOAT16 p</b>	The <b>FLOAT16</b> keyword, followed by a decimal position value, tells the driver that the next variable appearing in the argument list shall be filled from a half precision (16 bit) IEEE754 floating point value starting at the given position (position zero denotes the first byte in the buffer). The least significant byte of the number is read first unless <b>BIGENDIAN</b> was specified.

<b>FLOAT32 p</b>	The <code>Float32</code> keyword, followed by a decimal position value, tells the driver that the next variable appearing in the argument list shall be filled from a single precision (32 bit) IEEE754 floating point value starting at the given position (position zero denotes the first byte in the buffer). The least significant byte of the number is read first unless <code>BIGENDIAN</code> was specified.
<b>FLOAT64 p</b>	The <code>Float64</code> keyword, followed by a decimal position value, tells the driver that the next variable appearing in the argument list shall be filled from a double precision (64 bit) IEEE754 floating point value starting at the given position (position zero denotes the first byte in the buffer). The least significant byte of the number is read first unless <code>BIGENDIAN</code> was specified.
<b>XLT table</b>	The <code>XLT</code> keyword, followed by the name of an already defined table, tells the driver to translate the value read through this translation table from right to left. Chapter <a href="#">Using conversion tables</a> gives more information about tables in general.
<b>OFFSET o</b>	The <code>Offset</code> keyword, followed by the (floating point) offset value, tells the driver to add the given offset to the value read.
<b>SCALE s</b>	The <code>Scale</code> keyword, followed by the (floating point) scale factor, tells the driver to multiply value read with the given factor.
<b>FUNCTION "name"</b>	The <code>Function</code> keyword, followed by the name of the function as a quoted string, tells the driver to apply this function to the next variable before it is printed. The function name is extended by appending <code>.txt</code> to achieve the file name to read for the function definition. Chapter <a href="#">Function tables in I/O functions</a> gives an extensive description about functions and how they are used.
<b>BIGENDIAN</b>	The <code>BIGENDIAN</code> keyword turns the byte order from <i>little endian</i> (LSB first) which is the default to <i>big endian</i> (MSB first). This applies to all values read after <code>BIGENDIAN</code> .
<b>LITTLEENDIAN</b>	The <code>LITTLEENDIAN</code> keyword turns the byte order back to 'little endian' (LSB first). This applies to all values read after <code>LITTLEENDIAN</code> .
<b>variable-name</b>	A variable name tells the driver to store the value read into the variable.

### Example

```

READ BIGENDIAN
  INT32 1 SCALE 0.000001 tx.frequency
  INT32 7                tx.mod.dataRate
  INT32 11 SCALE 0.000001 refClkFreq
  INT8 15 XLT T01        refClkSrc
  INT8 16 XLT T02        tx.mod.type
  INT8 17 XLT T03        tx.mod.fec
  INT16 22 SCALE 0.1     tx.power
  INT8 24 XLT T04        tx.on
  INT8 24 XLT T04        internal.tx.on
  INT8 25 XLT T05        tx.mod.cwMode
  INT8 26 XLT T06        tx.mod.spectrumInvert
  INT8 28 XLT T14        tx.ifc.hardware
  INT8 29 XLT T06        tx.ifc.clockPhase
  INT8 30 XLT T06        tx.ifc.dataPhase
  INT8 31 XLT T07        tx.mod.clockSource
  INT8 44 XLT T08        info.maskEnable
  INT32 45               info.alarmMask
  INT32 49               tx.mod.symbolRate
  INT8 53 XLT T09        tx.ifc.framingMode
  INT8 54 XLT T10        tx.mod.rollOff
  INT8 55 XLT T11        config.control
  INT8 57 XLT T12        modemType

```

The example above - taken from the Radyne DVB3030 modulator driver - shows the main settings `READ` statement of this driver. In one step almost all parameter settings are read.

### 1.2.5.5 Function tables in I/O functions

The I/O functions of the ***sat-nms*** driver language (`INPUT`, `PRINT`, `READ`, `WRITE` and the REST I/O functions) all provide with the `FUNCTION` keyword a feature to translate a numeric value along a table of x/y value pairs. The result of the function is calculated by linear interpolation in this table.

#### File format

A `FUNCTION` requires the table of x/y values to be stored as a simple text file which looks like the following example:

```
# function table, defines a square root function
```

```
0.0  0.0  
1.0  1.0  
4.0  2.0  
9.0  3.0  
16.0 4.0  
25.0 5.0
```

The table consists of two or more lines, each consisting of the x-value followed by one or more space characters and the corresponding y-value. Both values must be valid floating point numbers. Blank lines and lines starting with a `#` character are ignored.

The x-values need not be equally spaced, but the values must be sorted in ascending order, starting with the lowest one.

### Calculation method

The function calculates the y-value to a given x-value by linear interpolation. This means, the curve defined by the x/y pairs is a polygonal line. The table should contain enough x/y-pairs to meet the expected accuracy of the function.

For x-values outside the range of x/y pairs defined in the table, the function extrapolates the y-value using the slope given by the outmost two x/y pairs. You may want to set two additional x/y-pairs outside the real table to set this slope to a particular value.

### File name processing

When executed in an I/O statement, the function extends the function name defined in the driver to the name of the file containing the function table. First the letters `.txt` are appended to the function name. The `.txt` file extension makes it easy to open the file with a text editor from the file manager of any operating system. Function table must be placed in the base directory of the **sat-nms** installation (usually `/home/satnms/`).

If the function name starts with a `-` character, the name of the device is prepended to the function name. For example, a function name `"-caltable"` would expand to a file name `DEVNAME-caltable.txt` where `DEVNAME` is the name of the actual device. This is useful for e.g. calibration tables which are different for each particular device.

### Error handling

The function checks if the file for this function exists. The driver raises a communication fault if a `FUNCTION` clause with non-existing file is processed. If the file itself cannot be parsed for some reason, the function tries to do its best, recovering from this situation if possible. A note is written to the `.panic.log` file, if such an error occurs.

## 1.2.6 REST I/O functions

Beside the basic I/O functions described in the previous chapters, the **sat-nms** driver definition

language provides some statements to handle the communication with devices using a REST API in an efficient way.

The REST API transports settings and status information as structured, JSON-formatted documents over an HTTP protocol. JSON documents are plain text and therefore may be - at least principally - handled by the basic PRINT / INPUT statements. In practice however, parsing a JSON document with the capabilities of INPUT may become difficult and in the end will result in a device driver which is hard to understand and to maintain.

The REST I/O statements contained in the **sat-nms** driver definition language take care of the special structure and formatting of JSON documents, helping to focus on the content to send or retrieve rather than the special JSON syntax. The statements for REST communication are:

<a href="#"><u>REST TRANSACTION</u></a>	Sends a REST HTTP request to the device, for HTTP <code>POST</code> , <code>PUT</code> or <code>PATCH</code> with a JSON formatted document body attached. Retrieves the device's reply and parses the document body into an object tree.
<a href="#"><u>REST PARSE</u></a>	Accesses the objects contained in the most recent reply from the device and assigns the object values to <b>sat-nms</b> driver variables.
<a href="#"><u>REST SET</u></a>	Modifies the objects contained in the JSON document or adds new objects to it. Used to prepare a document to be sent to the device with one of the <code>POST</code> , <code>PUT</code> or <code>PATCH</code> HTTP methods.
<a href="#"><u>REST CLEAR</u></a>	Clears the JSON document in memory. This function is used to prepare a document from scratch for a <code>POST</code> or <code>PATCH</code> call.
<a href="#"><u>REST SEND</u></a>	takes the actual JSON document and sends it to the device. It is used with protocols other than HTTP and replaces <code>REST TRANSACTION</code> there.

## Object model

A JSON document is a hierarchically structured collection of key / value pairs. The value in such a pair may be either a constant value, another ancillary collection or an array of these. The object model used in the **sat-nms** REST I/O functions flattens this tree-like structure, treats every constant value with its key as an object. The key of this constant value is extended to contain the complete information where in the document this key / value pair is located.

## Object paths

This location information can be thought as a path from the document's root down to the definition of the constant value of interest. This path is much like a path in a computer's file system. The higher-order nodes in the document tree are like directories in the file system. For this reason, the **sat-nms** driver definition language uses a notation like used with file names to identify an object in the document tree:

```
/settings/modulator/datarate
```

for example denotes an object `datarate` contained in the `modulator` branch of the

document which in turn is expected to be contained in the `settings` branch. The JSON equivalent of this would look like this:

```
{
  "settings": {
    "modulator": {
      "datarate": 2048000;
    };
  };
}
```

As with file names and paths, a path identifying an object may be absolute or relative. The example above uses an absolute path to the object: it starts with a slash character. When you access `/settings/modulator/datarate`, the software makes an implicit *change directory* to `/settings/modulator`. If you access then an object named for example `fec`, the software automatically will expand the object name to `/settings/modulator/fec`. This works inside a single `REST SET` or `REST PARSE` statement, not across separate statements.

## Object arrays

The array construct in a JSON document works in a special way and therefore needs some special treatment in the **sat-nms** driver definition language. JSON arrays are a collection of equally structured nodes. In most REST APIs, the elements of such a collection are not identified by their location in in the collection but by some unique identifier which appears as a regular key / value pair in each element of the array.

To cope with this construct, the **sat-nms** driver definition language extends the object path definition with "conditions" used to identify a particular collection element:

```
/settings/modulator/output[id=3]/enable
```

addresses the `enable` object in that `output` branch that has a key `id` set to `3`. Conditions may appear after each branch name in the object path. There may be multiple conditions which are combined with a logical **and**:

```
/settings/modulator/connector[number=2][type=OUTPUT]/enable
```

addresses the `enable` object in that `connector` branch that has a key `number` set to `3` **and** `type` set to `OUTPUT`.

Alternatively, array elements may be addressed by their position in the array. The numeric array index starts with `0` for the first array element:

```
/settings/modulator/output[2]/enable
```

addresses the `enable` object in the third (index=2) 'output' branch. Please note, that this only works, if the API of the device to control explicitly permits to access array elements by position.



Furthermore it is not possible to create array elements by position index in the `REST SET` statement.

## Document handling

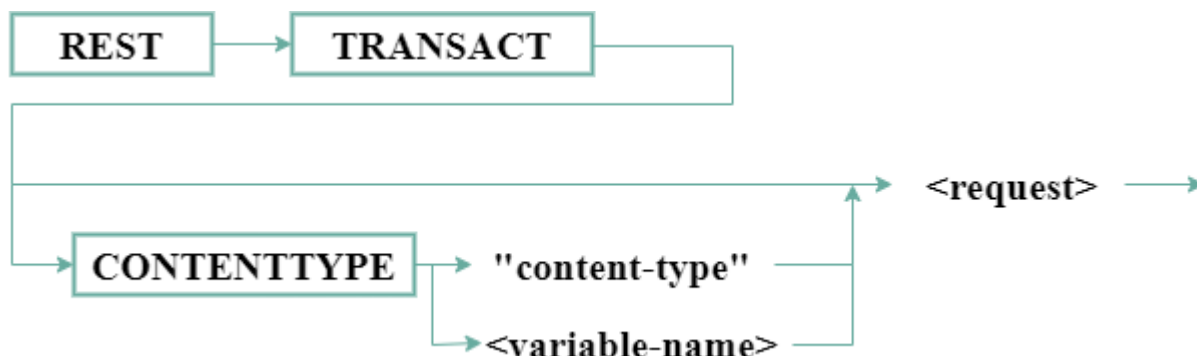
With the REST statements in the **sat-nms** driver definition language there is always one JSON document in memory these statements work on. Generally, it is the `REST TRANSACT` statement which creates this document from the device's reply, `REST CLEAR` creates an empty document instead. The `REST SET` statement can be used to add or modify objects in the document, the `REST PARSE` statement to query objects from there.

With a `REST TRANSACT` doing a HTTP `GET` command you can expect to get the information you addressed in the `GET` command, but `POST`, `PUT` or `PATCH` also may return a document containing some information that might be important for the device driver.

When issuing a `POST`, `PUT` or `PATCH` command, the `REST TRANSACT` statement sends the existing document to the device using the selected HTTP method, then replaces the document by the parsed reply from the device.

### 1.2.6.1 The REST TRANSACT statement

The `REST TRANSACT` statement sends a REST HTTP request to the device, either with or without a JSON formatted document body. It retrieves the device's reply and parses the document body into an object tree.



The request is an arbitrary long sequence of quoted strings and **sat-nms** variables. Strings and variable content are concatenated to build the request string:



The request must start with the HTTP method to execute (one of `GET`, `PUT`, `POST`, `PATCH` or `DELETE`), followed by one space character and the path of the URL to read. The path may contain URL parameters. Example:

```
GET /api/v1/settings/modulator?slot=2
```

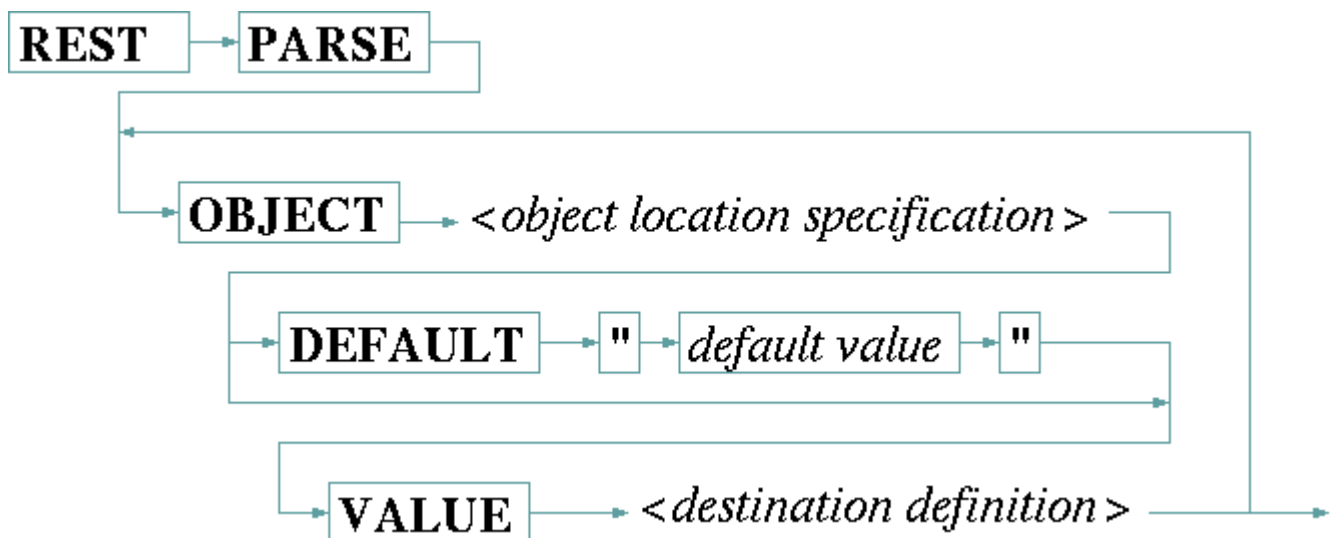
Please note, that no URL-encoding is done on this string, this means that all special character must be escaped (e.g. blanc replaced by '+')

With the **POST** , **PUT** or **PATCH** HTTP methods, **REST TRANSACT** sends the JSON document actually in memory to the device. This document may be created from scratch or the recent reply from the device may be used as a template which is modified before the transaction is executed. To transact a HTTP method other than GET without sending a JSON document, place a **REST CLEAR** statement directly above the **REST TRANSACT** .

**REST TRANSACT** may be immediately followed by the keyword **CONTENTTYPE** and a content type definition in double quotes or contained in a variable. This content type definition overrides any **PROTOCOLPARAMETER** `post.content.type=...` definition made in the protocol or driver header.

### 1.2.6.2 The REST PARSE statement

The **REST PARSE** statement accesses the objects contained in the most recent reply from the device and assigns the object values to **sat-nms** driver variables. It uses the same object model as described above for the **REST TRANSACT** statement to address and extract the desired information from the JSON document the device replied through its REST API.

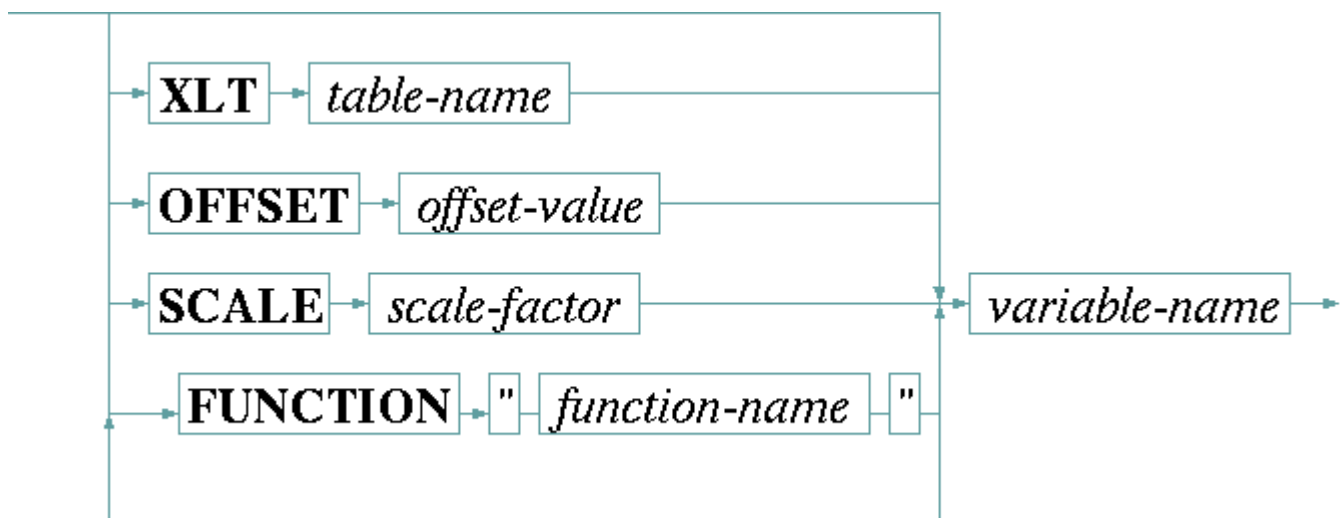


The **OBJECT** keyword followed by the object location specification addresses the object in the JSON tree to be read. The object location specification works exactly as described above with one difference: non-existent objects are not created but treated as a fault unless there is a default value defined for this object.

The optional **DEFAULT** keyword denotes the default value to be assigned if the specified object does not exist in the object tree. The default value is given as a quoted string following the **DEFAULT** keyword. Please note, this value is treated as value read from the JSON document, it undergoes the modifications like **XLT** , **SCALE** or **OFFSET** following **VALUE**

before it is assigned to the **sat-nms** driver variable. If no **DEFAULT** is defined within an **OBJECT** clause and the addressed object is missing in the JSON document, the driver will raise a communication fault.

The keyword **VALUE** starts the definition of the **sat-nms** driver variable which shall receive the extracted value together with some modifiers known from the **INPUT** statement. The destination definition in any case ends with then name of the **sat-nms** driver variable which may be prepended by zero or more of the modifiers **XLT**, **OFFSET** or **SCALE** as shown below:



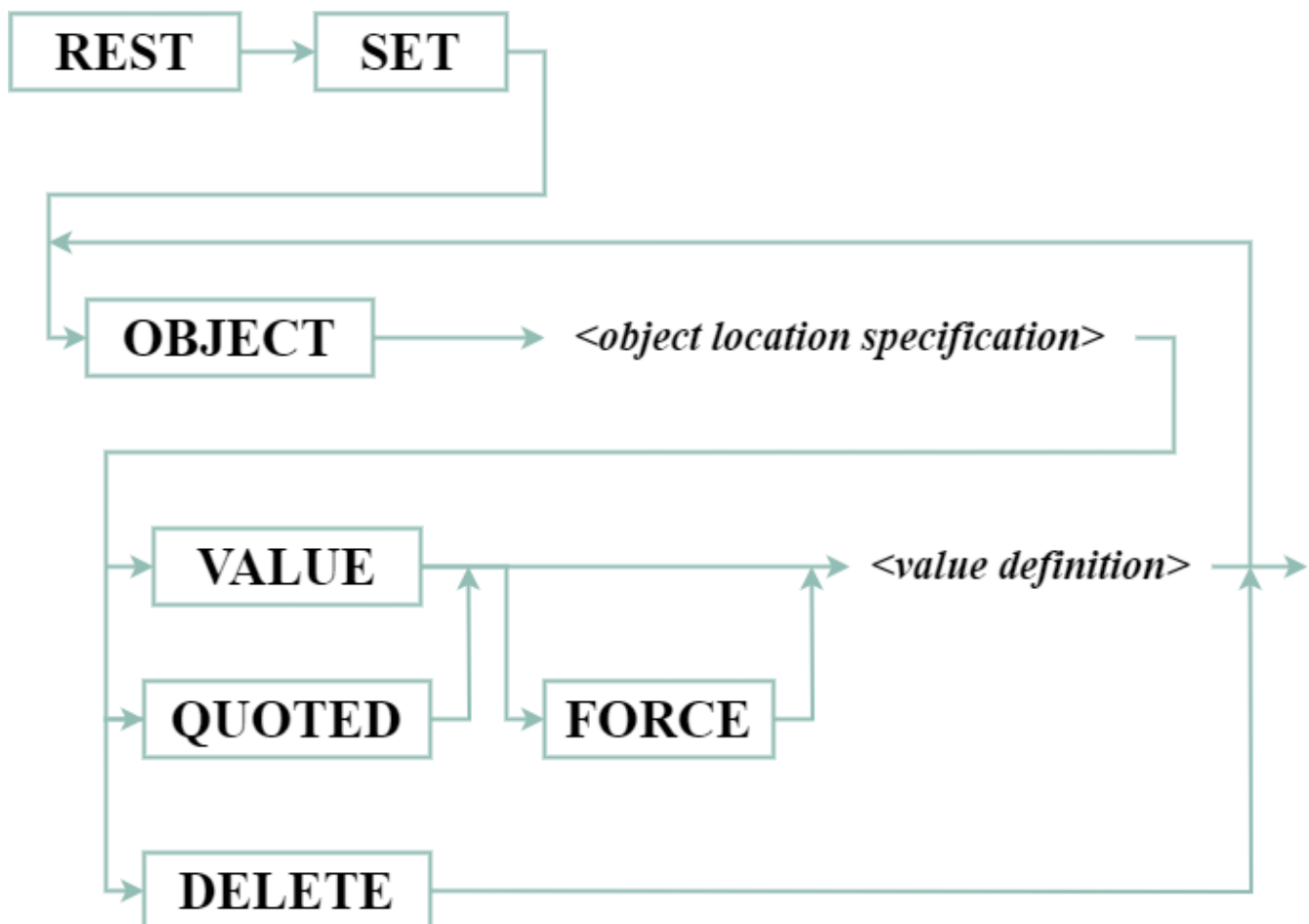
<b>XLT table</b>	The <b>XLT</b> keyword, followed by the name of an already defined table, tells the driver to translate the object's value through this translation table from right to left. Chapter <a href="#">Using conversion tables</a> gives more information about tables in general.
<b>OFFSET o</b>	The <b>OFFSET</b> keyword, followed by the (floating point) offset value, interprets the object's value as a floating point number and replaces it with the sum of the value and the offset.
<b>SCALE s</b>	The <b>SCALE</b> keyword, followed by the (floating point) scale factor, interprets the object's value as a floating point number and replaces it with the product of the value and the scale factor.
<b>FUNCTION "name"</b>	The <b>FUNCTION</b> keyword, followed by the name of the function as a quoted string, tells the driver to apply this function to the next variable before it is printed. The function name is extended by appending <b>.txt</b> to achieve the file name to read for the function definition. Chapter <a href="#">Function tables in I/O functions</a> gives an extensive description about functions and how they are used.

If there are multiple driver variables to be read from a single REST reply sent by the device, you may either use separate **REST PARSE** statements for this or use a single statement with multiple **OBJECT** clauses. Example

```
REST TRANSACT "GET /api/v1/modulators"
REST PARSE
  OBJECT "/settings/modulator[slot=" config.slotNo "]/frequency"
    DEFAULT "1500.000" VALUE tx.frequency
  OBJECT "/settings/modulator[slot=" config.slotNo "]/fec"
    DEFAULT "3_4" VALUE XLT tFec tx.fec
```

### 1.2.6.3 The REST SET statement

The **REST SET** statement modified objects in the actual JSON document tree or adds new key / value pairs to the document. The statement starts with the REST SET keywords followed by one or more **OBJECT ... VALUE** or **OBJECT ... QUOTED** clauses:

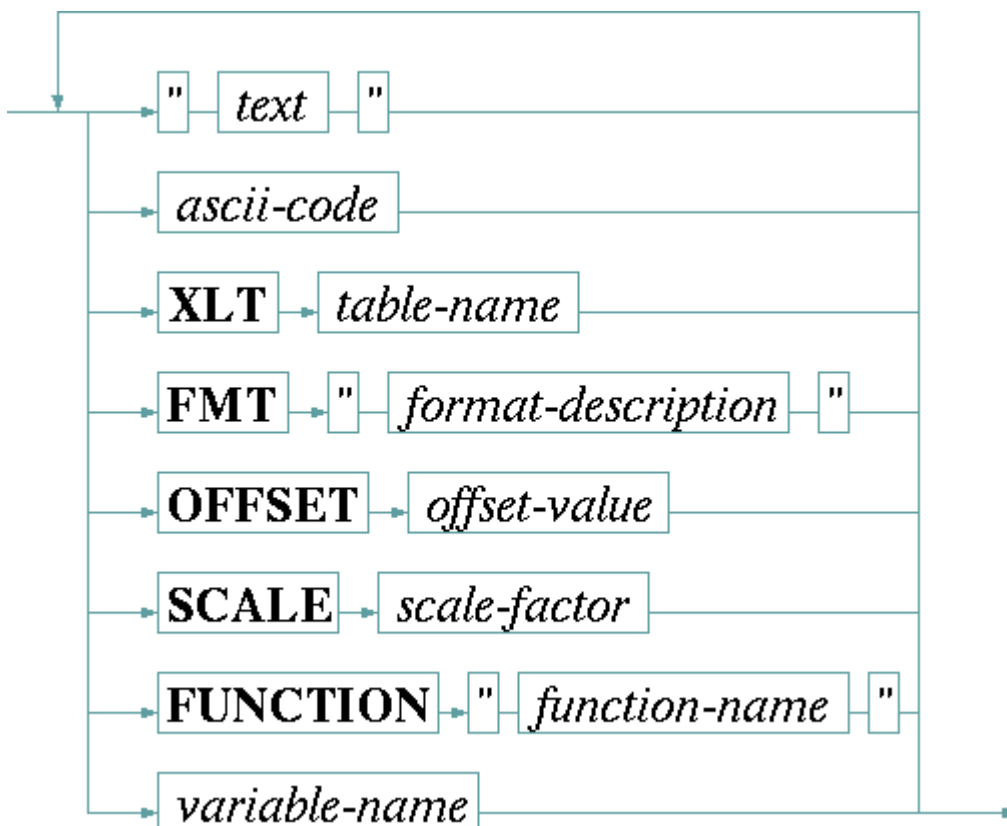


Each object definition starts with the **OBJECT** keyword, followed by the object location definition. The object location definition may be concatenated from multiple quoted strings and **sat-nms** driver variables. Section *object paths* in chapter [REST I/O functions](#) describes object location definitions more detailed.



Within the **REST SET** statement, each object is newly created with all its parent objects if it does not yet exist in the document. This includes *conditional* definitions for array elements. If an object already exists in the document, its value will be overwritten by the value specified later in this **OBJECT** clause.

The object location definition is terminated by the **VALUE** keyword which starts the definition of the value which shall be assigned to this object. Alternatively, **QUOTED** may be used instead of **VALUE**, this is used for textual values which shall be enclosed in double quotes in the JSON text.



The value definition itself is much like writing a command using the **PRINT** command:

<b>"text"</b>	Plain text, enclosed in double quotes, is added to the output buffer as it is.
<b>ascii-code</b>	A decimal number is interpreted as the ASCII code of a single character to output. You may use this to send special characters like carriage-return or line-feed to the device.

<b>XLT table</b>	The <code>XLT</code> keyword, followed by the name of an already defined table, tells the driver to translate the next variable value which shall be printed through this translation table. Chapter <a href="#">Using conversion tables</a> gives more information about tables in general.
<b>FMT "..."</b>	The <code>FMT</code> keyword, followed by a format description in double quotes, tells the device driver to format the next variable following the format description given. <code>FMT</code> is used to format numeric variables into the representation the device expects in it's remote control command.

A format description consists of the following elements:

format character	description
<b>d b x X f</b>	The first letter of the format description defines the general number representation: <code>d</code> (decimal), <code>b</code> (binary), <code>x</code> (hex, lower case), <code>X</code> (hex, upper case) or <code>f</code> (floating point)
<b>+</b>	If the <code>+</code> option is given, the number is preceded by a +/- character even if it is positive. Together with the <code>0</code> option below, the sign appears as an additional character at the first column, hence the field width is increased by one on this case.
<b>0</b>	If the <code>0</code> option is given, the field is padded up with zeroes instead of spaces.
<b>1 .. 99</b>	Here follows a one or two digit field width. The field width is the total number of characters the formatted number occupies including the padding characters. If there are more digits needed to show a number correctly, the field with is enlarged automatically. Specifying a field width <code>1</code> disables the right orientation in a fixed width completely. This specially is useful for floating point numbers where only the number of fraction digits shall be fixed, not the complete field width.
<b>.</b>	For floating point formats the dot separates the precision from the field width.
<b>0 .. 9</b>	The dot is followed by a one digit specification of the number of fraction digits which shall be printed. The dot and fraction digits specification is valid only with the floating point format.

`INTEGER` variables may be printed using a floating point format, and `FLOAT` variables may be printed with a `d` or `x` format likewise. With `FLOAT` variables you should format in any case as internal precision/rounding problems may cause unpredictable results when floating point values are printed unformatted.

<b>OFFSET o</b>	The <code>OFFSET</code> keyword, followed by the (floating point) offset value, tells the driver to add the offset value to the next variable before it is printed.
<b>SCALE s</b>	The <code>SCALE</code> keyword, followed by the (floating point) scale factor, tells the driver to multiply the next variable with the scale factor before it is printed
<b>FUNCTION "name"</b>	The <code>FUNCTION</code> keyword, followed by the name of the function as a quoted string, tells the driver to apply this function to the next variable before it is printed. The function name is extended by appending <code>.txt</code> to achieve the file name to read for the function definition. Chapter <a href="#">Function tables in I/O functions</a> gives an extensive description about functions and how they are used.
<b>variable-name</b>	A variable name tells the driver to print the value stored in this variable. Usually the <i>commanded</i> value is used rather than the value which has been read back from the device. If, however, there has been never a value commanded, or if this variable is a read only variable, the value recently read from the device is used.

Before the variable is printed to the output buffer, any `XLT` , `FMT` , `OFFSET` or `SCALE` operations which have been specified are applied. This happens in a fixed order:

1. If a `SCALE` operation has been specified, this is done first.
2. The `SCALE` operation is followed by an `OFFSET` addition.
3. The so scaled value is formatted according to a `FMT` specification if one is given.
4. Finally, the value gets translated through a translation table, if an `XLT` operation has been specified.

This scheme applies to numeric ( `INTEGER` , `HEX` , `FLOAT` ) and to text type variables as well. As soon as a `SCALE` , `OFFSET` or `FMT` keyword is present in front of a variable, this gets converted to a floating point number. Strings which cannot be converted give a zero value.

## Example

```
REST CLEAR
REST SET OBJECT "/settings/frequency" VALUE FMT "f1.3" tx.frequency
REST SET OBJECT "/settings/fec" QUOTED XLT "tFec" tx.fec
TRANSACT "PATCH /api/v1/modulators?slot=2"
```

## Value quoting

In JSON, textual values are enclosed in double quotes, numeric and boolean values are not. The ***sat-nms*** Software does this automatically by looking at the value to set if this is a boolean, a number or a text. This may fail, if the device expects a number enclosed in double quotes.

To cope with this, the `REST SET` statement lets you force the usage of quotes in several ways:

1. Enclose the value in quotes when sending it. You may use single quotes instead of double quotes to avoid the need of escaping the double quote with a backslash. The JSON document sent to the device will contain double quotes instead of the single ones used in the driver. With constraint definitions in the object path this is the only way to force quotes if required.
2. For the value set in a JSON object, you may use the `QUOTED` keyword instead of `VALUE`. The value will be forced to be set in double quotes when added to the document. This way produces better readable code than putting the quote characters explicitly in the sequence of operands after `VALUE`.

### Conditional execution

By default a `VALUE` or `QUOTED` clause in the `REST SET` statement causes the JSON document only to be modified if

- The value to be set is a constant
- The variable to be set is marked *asto be written* by the driver. This is if the driver was commanded to set this value, but this did not yet happen.

This behavior protects the JSON document content from unwanted changes if e.g. only one parameter shall be changed but the `REST SET` statement contains the `OBJECT .. VALUE` pairs of all parameters in this group.

You can force values to be written regardless of the conditions described above by introducing the keyword `FORCE` directly after the `VALUE` or `QUOTED` keywords.

### Object deletion

When used with `DELETE`, `REST SET` deletes the JSON object specified in the `OBJECT` clause and all of its child objects.

Parent objects are removed as well if they become empty after deletion of the specified object. Please note, that array elements may still not be empty - even if all known child elements of the array element have been deleted. In this case the array element is not automatically removed, must be deleted explicitly by another `REST SET ... DELETE` statement.

#### 1.2.6.4 The REST CLEAR statement

The `REST CLEAR` statement creates an empty JSON document in memory, which deletes the document left by the recent `REST TRANSACT` statement.



`REST CLEAR` is commonly used to prepare a JSON document from scratch with the `POST` or `PATCH` HTTP methods.

#### 1.2.6.5 The REST SEND statement

The `REST SEND` statement takes the actual JSON document and sends it to the device. It is



used with protocols other than HTTP and replaces `REST TRANSMIT` there.

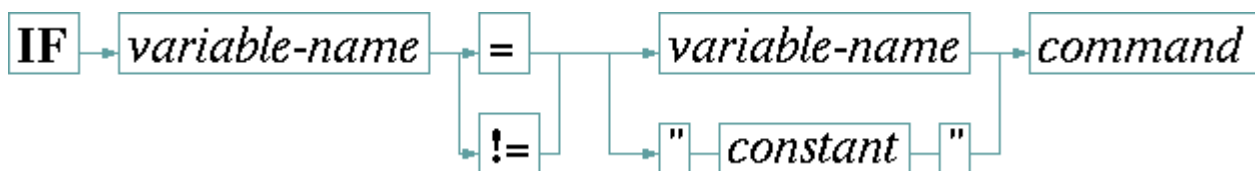


## 1.2.7 Conditional execution

The device driver language defines a couple of statements to control the flow of execution within a procedure. There is an `IF` statement as well as a `GOTO` which refers to a `label` placed elsewhere in the procedure.

### 1.2.7.1 The IF statement

The `IF` statement conditionally executes a single statement if the comparison following the `IF` keyword is true.



The keyword `IF` is followed by a comparison of a variable to another variable or constant. With the `=` operator the command is executed if the compared values match, with `!=` if they don't match respectively. Please note, that a constant value must be enclosed in double quotes, even if it is a numeric value. `IF` compares the string representation of the variables after formatting them as defined in the `VAR` statement. Hence, floating point variables always are compared after being rounded to the number of digits defined for the user interface display.

### Example

```

IF info.version = "2.0" PRINT "TFR " FMT "f1.3" tx.frequency
IF info.version != "2.0" PRINT "TFR " FMT "d08" SCALE 1000.0 tx.frequency
  
```

Depending on the value of the `info.version` variable, the example above uses different commands to set the `tx.frequency` at the device. `PRINT` is considered as a single command, with all its parameters in this case. If more than one command in sequence shall be executed conditionally, a `GOTO` statement must be used to jump over the this command sequence.

### 1.2.7.2 The GOTO statement

The `GOTO` statement branches to another location in the same procedure. The destination where to branch to is referenced by a `label`. The `GOTO` statement lets the execution of the procedure continue at the statement following the label.



The label, the `GOTO` statement refers to may be defined above or below the location of the `GOTO` statement. However, the label must be defined in the same procedure. The device driver interpreter does not support branches across procedure boundaries.

### Some words about loops

Principally `GOTO` statements may be used to create loops within a procedure. You are strongly discouraged from doing so. This is because all procedures for a device (more exactly spoken the procedures of all devices operated at the same physical interface) are executed sequentially. If now one procedure is caught in a loop for some time or perhaps forever, no other procedures will run. The device driver will appear to be *frozen*.

#### 1.2.7.3 Label definition

Labels are defined as destination locations for the `GOTO` statement. The colon, followed by a label name defines a label:

`:` → *label-name*

Labels only are visible within the procedure where they are defined. As a consequence, you may define labels with the same name in different procedures. Within one procedure however, a label name may be used only once. Between the colon and the label name no whitespace is required.

#### Example

```
IF info.version = "2.0" GOTO v2stuff
  PRINT "TFR " FMT "d08" SCALE 1000.0 tx.frequency
  GOTO finished
:v2stuff
  PRINT "TFR " FMT "f1.3" tx.frequency
:finished
```

#### 1.2.8 Using subroutines

Procedures may be defined to act as subroutines (see chapter [The PROC statement](#) for the syntax description). They are not bound to variables in this case. The `CALL` statement is used to invoke a subroutine from another procedure.

Subroutines are suitable to code operations which have to be done in the same way in several procedures. A common application for this is the check of the device's reply to a command. A subroutine can read the device's reply, check if it is OK and raise a fault flag if not. After a procedure sends a command to the device, it calls the subroutine which does the check.

When using subroutines, you should notice the following:

- Subroutines may be nested, this means a subroutine may contain a `CALL` statement to another subroutine.

- Subroutines must be defined before they can be referenced in another procedure.
- The device driver language allows to define recursive subroutines (subroutines which call themselves). You are strongly discouraged to program recursive subroutines as a program bug in your device driver may cause infinite recursion which definitely will crash the whole MNC application!

### 1.2.8.1 The CALL statement

The `CALL` statement is used to invoke a subroutine from another procedure. The called routine must be defined before it can be called.

`CALL` → *subroutine-name*

The driver executes all statements of the called procedure and then resumes

### Examples

The example below, taken from the Tandberg-Alteia device driver, shows a common application of a subroutine. The subroutine `getAck` is called every time the driver sent a command to the IRD to check if the unit accepted the command.

```
// called after sending a command to the IRD. checks the ACK response which is
// expected.
//
PROC SUBROUTINE getAck
  INPUT "=" TRM "_" internal.ack
  IF internal.ack = "ACK" GOTO endif
    SET faults.commstat = "ACK expected"
    SET faults.99 = "true"
:endif
```

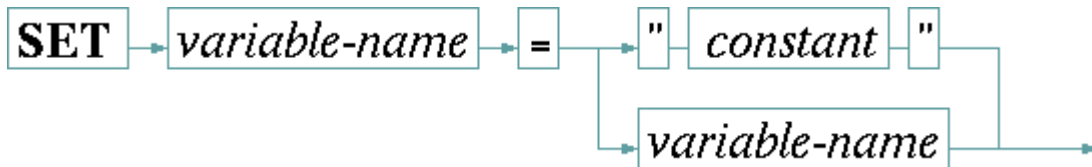
### 1.2.9 Manipulating variables

The **sat-nms** device driver definition language provides a number statements to manipulate driver variables within procedures. These are:

<a href="#">SET</a>	Assigns a value to a variable.
<a href="#">BITSET</a>	Extracts a single bit from a variable and assigns it to another.
<a href="#">RANGESET</a>	Changes the range definition of a variable.
<a href="#">BITSPLIT</a>	Splits up a variable in single bits. Should no longer be used.
<a href="#">BITMERGE</a>	Merges several variables each containing one bit to another variable. Should no longer be used.

### 1.2.9.1 The SET statement

The **SET** statement lets you assign a value to a variable. The value may be a constant in double quotes (even numeric values must be quoted) or the contents of another variable.



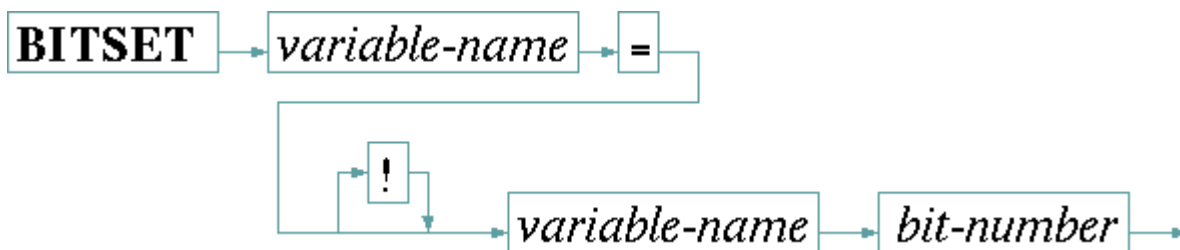
**SET** assigns the value to the internal memory in the variable which remembers the value read from the device. Assigning a value with set hence has the same effect as assigning it within a INPUT or READ statement. **SET** does not change the commanded value a variable receives from the user interface.

#### Example

```
SET tx.power = "33.2"
```

### 1.2.9.2 The BITSET statement

Some devices report their fault state as a number in which each bit represents one fault flag. The **BITSET** statement is used to decode flags from such a status value. Usually the fault status is read into an internal variable if the driver. Then the fault bits are decoded from this internal variable using the **BITSET** statement.



The bit position zero addresses the least significant bit in the source variable. The destination variable gets set to **1** or **true** if the addressed bit is set, to **0** or **false** if not. If the variable name is preceded by an exclamation mark, this mapping is inverted.

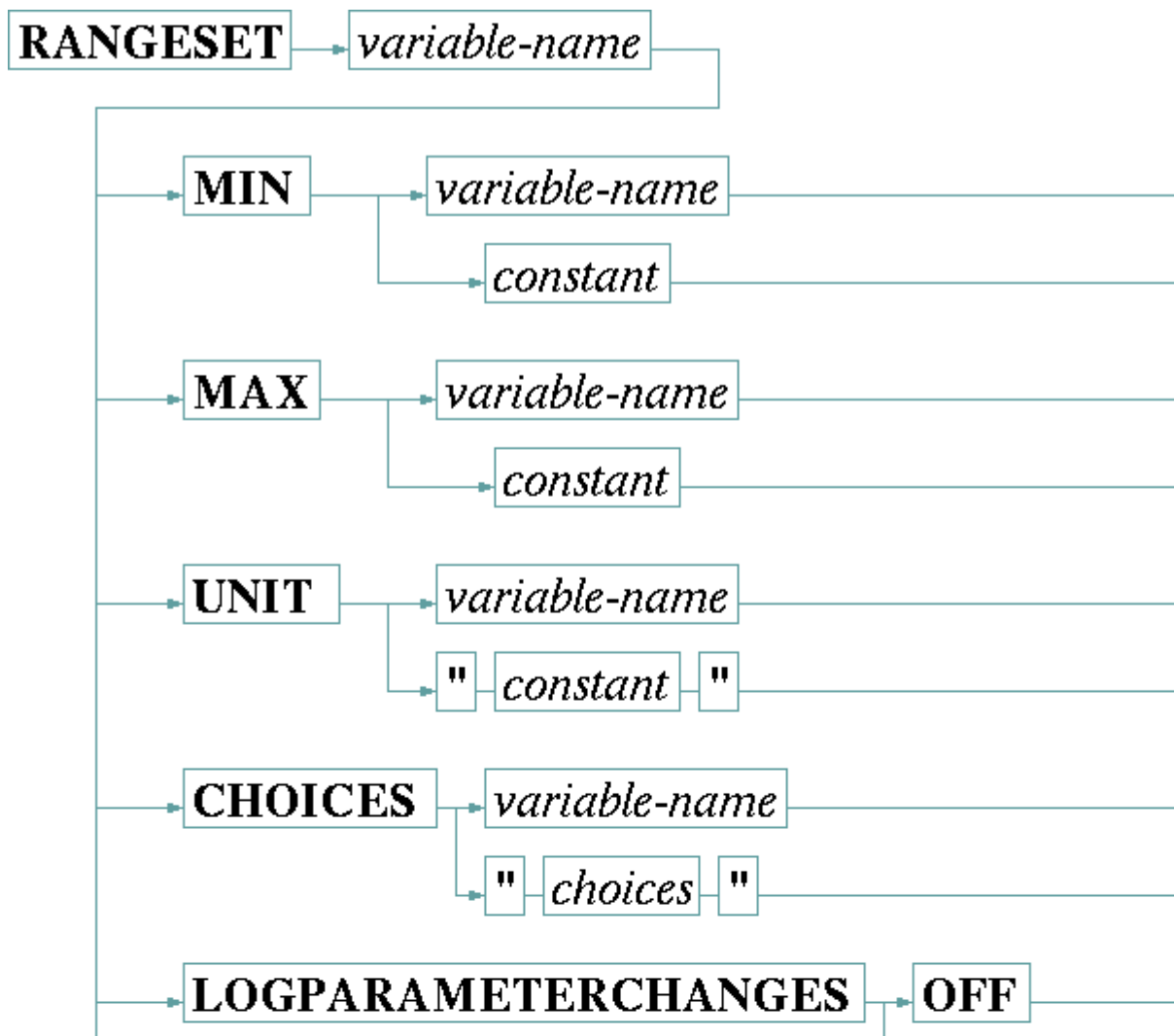
#### Example

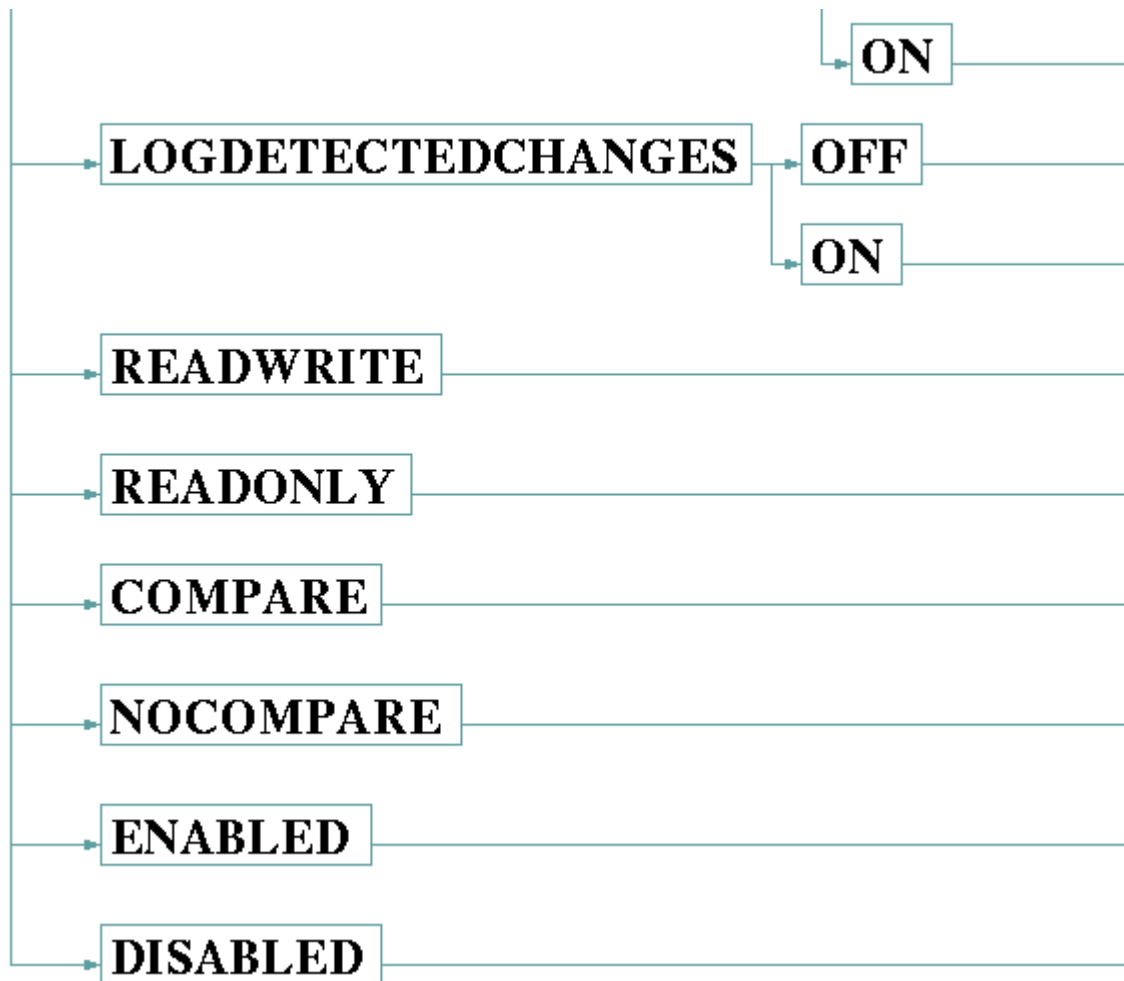
```
INPUT internal.flags
BITSET faults.01 = internal.flags 4 // Temperature
BITSET faults.02 = internal.flags 5 // Signal level
BITSET faults.03 = ! internal.flags 11 // Video lock (inv)
BITSET faults.04 = ! internal.flags 9 // Audio lock (inv)
BITSET faults.05 = internal.flags 13 // High BER
BITSET faults.06 = internal.flags 15 // Demodulator lock
BITSET faults.07 = internal.flags 14 // Conditional Access
```

The example above reads a value into the variable `internal.flags`. BITSET statements are used to extract seven faults flags from this variable. `internal.flags` must be declared `READONLY` to make this example work.

### 1.2.9.3 The RANGESET statement

The `RANGESET` statement changes several aspects of the range definition of a variable. It is intended to be used in drivers which read some capabilities (e.g. a frequency range) from the device itself.





The options which may follow the `RANGESSET` statement are described in the table below:

<b>MIN</b>	Changes the minimum value for a numeric ( <code>INTEGER</code> , <code>HEX</code> , <code>FLOAT</code> ) variable. The new minimum value may be a constant numeric value or the contents of another variable. <code>MIN</code> applies to numeric variables only.
<b>MAX</b>	Changes the maximum value for a numeric ( <code>INTEGER</code> , <code>HEX</code> , <code>FLOAT</code> ) variable. The new maximum value may be a constant numeric value or the contents of another variable. <code>MAX</code> applies to numeric variables only.

<b>UNIT</b>	Changes the maximum value for a numeric ( <code>INTEGER</code> , <code>HEX</code> , <code>FLOAT</code> ) or a <code>ALARM</code> flag variable. The new unit string may be a constant value enclosed in double quotes or the contents of another variable. <code>RANGESET</code> / <code>UNIT</code> applied to <code>ALARM</code> flags changes the alarm description text. To make the unit string for a numeric variable look like the unit specified in the <code>VAR</code> statement, start the string with a space character (e.g. <code>dBm</code> ). The <code>VAR</code> statement implicitly prepends this space character.
<b>CHOICES</b>	Changes the set of choices for a <code>CHOICE</code> variable. May be followed either by a variable name or by a quoted string defining the new list of choices. The list of choices is a comma separated list of strings, either defined as a constant enclosed in double quotes or read from the contents of the variable referenced after the <code>CHOICES</code> keyword. <code>CHOICES</code> applies to <code>CHOICE</code> variables only.
<b>READWRITE</b>	Makes the parameter writable from the user interface, overrides a former <code>READONLY</code> definition. <code>READWRITE</code> applies to all types of variables.
<b>READONLY</b>	Makes the parameter read only. <code>READONLY</code> applies to all types of variables.
<b>LOGPARAMETERCHANGESON/OFF</b>	Defines if this variables shall log changes which are commanded to it. Parameter changes are logged if this is set ON and the <code>logParameterChanges</code> switch in the device setup page is activated as well. <code>LOGPARAMETERCHANGES</code> is <code>ON</code> by default for all variables, hence you only need to state <code>LOGPARAMETERCHANGES OFF</code> for variables you want explicitly to be excluded from logging.

<b>LOGDETECTEDCHANGESON/OFF</b>	<p>Defines if this variables shall log changes which are detected when reading back the values. Detected changes are logged if this is set <b>ON</b> and the <b>logDetectedChanges</b> switch in the device setup page is activated as well. <b>LOGDETECTEDCHANGES</b> is <b>ON</b> by default for all variables, hence you only need to state <b>LOGDETECTEDCHANGES OFF</b> for variables you want explicitly to be excluded from logging. Please note, the this modifier has no effect on read-only variables, they are not logged regardless of the logDetected setting.</p>
<b>COMPARE</b>	<p>Enables the read after write comparison check for this variable. This check generates a log message, if the value of the variable read back from the device differs from the commanded value. By default the read after write comparison check is enabled for all variables unless they are defined with the <b>NOCOMPARE</b> option in the VAR statement.</p>
<b>NOCOMPARE</b>	<p>Disables the read after write comparison check for this variable.</p>
<b>ENABLED</b>	<p>Enables a formerly disabled variable. <b>ENABLED</b> applies to all types of variables.</p>
<b>DISABLED</b>	<p>Disables a variable. Disabled variables show no value at the user interface, any user input is blocked. Disabled variables never cause a procedure which watches this variable to run. <b>DISABLED</b> applies to all types of variables.</p>

Updating the range of a variable marks the variable to be read from the device as soon as possible.

The **sat-nms** user interface recognizes range definitions every time a window is opened. An already opened window does not change the range definitions of it's input elements. A driver should change variable ranges only due to information which is read once on power up or after a device has been switched on.

### Example



```

PRINT "RMAF" INPUT "=" internal.rx.fmax
PRINT "RMF" INPUT "=" internal.rx.fmin

RANGESET rx.frequency MAX internal.rx.fmax
RANGESET rx.frequency MIN internal.rx.fmin

```

The example above - taken from the SSE K-Star device driver - reads the valid frequency range for the device into the internal driver variables `internal.rx.fmax` and `internal.rx.fmin`.

`RANGESET` statements then update the range definition for the `rx.frequency`. This is done in a procedure which is called once after the driver gains communication to the device.

#### 1.2.9.4 The BITSPLIT statement

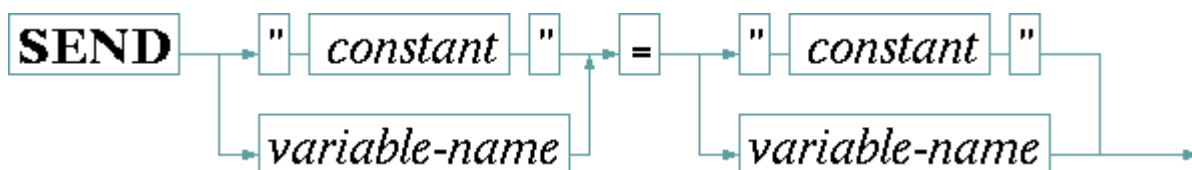
The `BITSPLIT` statement splits up a numeric variable in single bits. `BITSPLIT` will no longer be supported by future versions of the software and therefore should not be used in new device drivers.

#### 1.2.9.5 The BITMERGE statement

The `BITMERGE` statement sets a numeric variable from a set of other variables each controlling one bit in the destination variable. `BITMERGE` will no longer be supported by future versions of the software and therefore should not be used in new device drivers.

#### 1.2.9.6 The SEND statement

The `SEND` statement builds a parameter message from an address and value part and sends this to another device for execution, just like if the parameter had been issued by an operator of the software. The main purpose of this statement is to build logical devices interacting between other device by means of the universal driver language.



The syntax of the `SEND` statement looks much like the SET statement: Following the `SEND` keyword, you specify the destination and the value to be set, separated by `=` character. Both, the destination definition and the value to be sent, may either be quoted string or local variables of this driver:

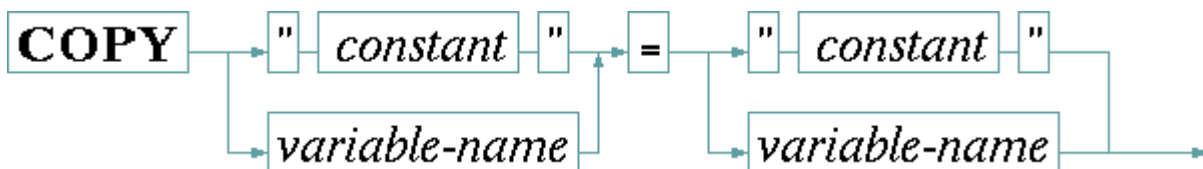
Quoted String	Variable
---------------	----------

Quoted String	Variable
<b>Destination</b>	The message destination is literally specified like <code>DEVICE1.parameterName</code> . The contents of the variable (e.g. <code>config.destinationId</code> ) is used as the destination ID for the parameter message. The variable must contain a string value representing a valid message ID
<b>Value</b>	The value to be set at the destination parameter is literally specified. Please note, that also numeric values must be enclosed in quotes here to be recognized as a literal constant. The contents of the variable is send as the value of the parameter message.

At execution time, the `SEND` statement checks if the destination parameter exists and converts the type of the value to be set as required. If this fails, you will find a `SEND to xxxx failed` informational message in the event log in this case.

### 1.2.9.7 The COPY statement

The `COPY` statement copies either a single parameter or the complete setting from one device to another

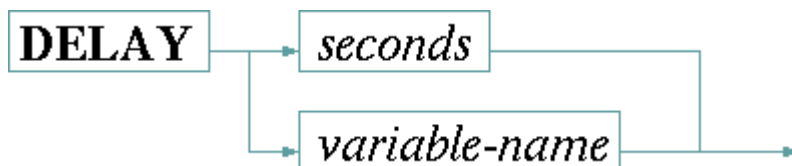


The **sat-nms** device driver definition language contains some more statements which have not yet been described in this document. They are:

<a href="#"><u>DELAY</u></a>	Pauses the device driver execution for some time.
<a href="#"><u>LOG</u></a>	Adds a message to the event log.
<a href="#"><u>DRATE</u></a>	Computes an interface data rate from a symbol rate.
<a href="#"><u>SRATE</u></a>	Computes a symbol rate from an interface data rate.
<a href="#"><u>WRITEHEX</u></a>	Outputs a variable containing a hex dump string as binary data.
<a href="#"><u>READHEX</u></a>	Reads binary data from the device and formats this as a hex dump string into a variable.
<a href="#"><u>INVALIDATE</u></a>	Invalidates a variable, marks it to be read back as soon as possible.
<a href="#"><u>SYNC</u></a>	Syncs a variable, updates the value to be commanded with the value recently read back from the device.
<a href="#"><u>RAISECOMMFAULT</u></a>	Makes the driver to raise a communication fault.

#### 1.2.10.1 The DELAY statement

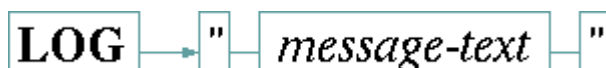
The **DELAY** statement pauses the driver execution for a given number of seconds. The delay may be specified as a (floating point) constant or a variable may be referenced which contains the time to delay.



You should use **DELAY** statements with care. Delaying the driver execution not only slows down the execution of the current procedure, it delays the polling cycle of all devices operated at this serial interface. With longer delay times, the MNC system may no longer be able to recognize device faults within a reasonable time.

#### 1.2.10.2 The LOG statement

The **LOG** statement lets you add an arbitrary message to the MNC system's event log. This may be used to signal events which shall not be treated as a fault, but are to be recorded anyhow.



Messages generated by the **LOG** statement by default are of the priority **INFO**. If the first

character of the message to send is `2` or `3`, the first character is removed from the message and the priority id increased to `FAULT` (2), `ALARM` (3) or `WARNING` (4) respectively.

### Example

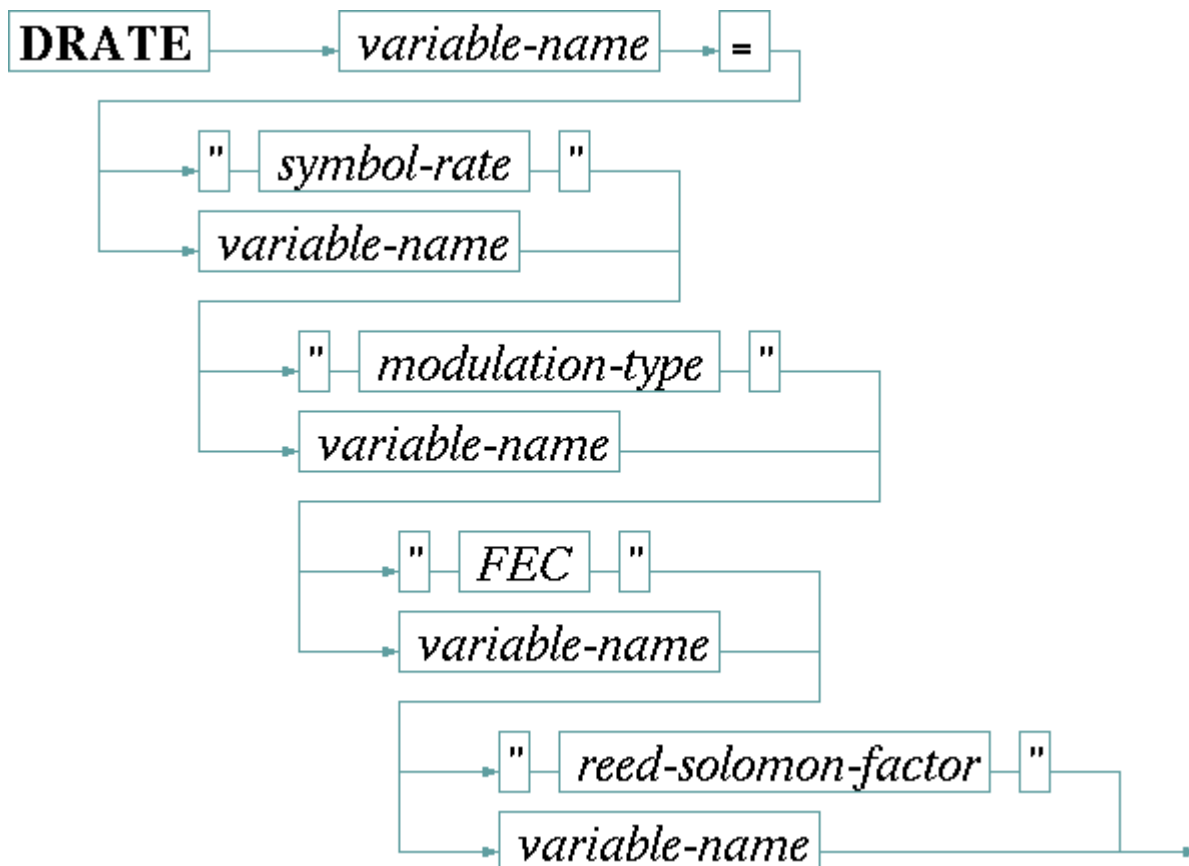
```
LOG "This is an informational message"
LOG "3This is an ALARM message !"
```

The example above adds two messages to the log. The message `This is an informational message` is added with priority `INFO`, the message `This is an ALARM message!` is added with `ALARM` priority.

Please note: although the priority `WARNING` is represented by the number 4, it's logical priority is between `INFORMATIONAL` and `FAULT`. This is because warnings have been added to the software lately and there had to be defined an unused number for the new priority.

### 1.2.10.3 The DRATE statement

The `DRATE` statement implements a data rate calculator which lets you convert a symbol rate to a interface data rate. This may be useful if a device driver shall supply both values, but the device itself only supports one of these settings.



The `DRATE` statement collects a symbol rate value, a modulation type, a FEC value and a

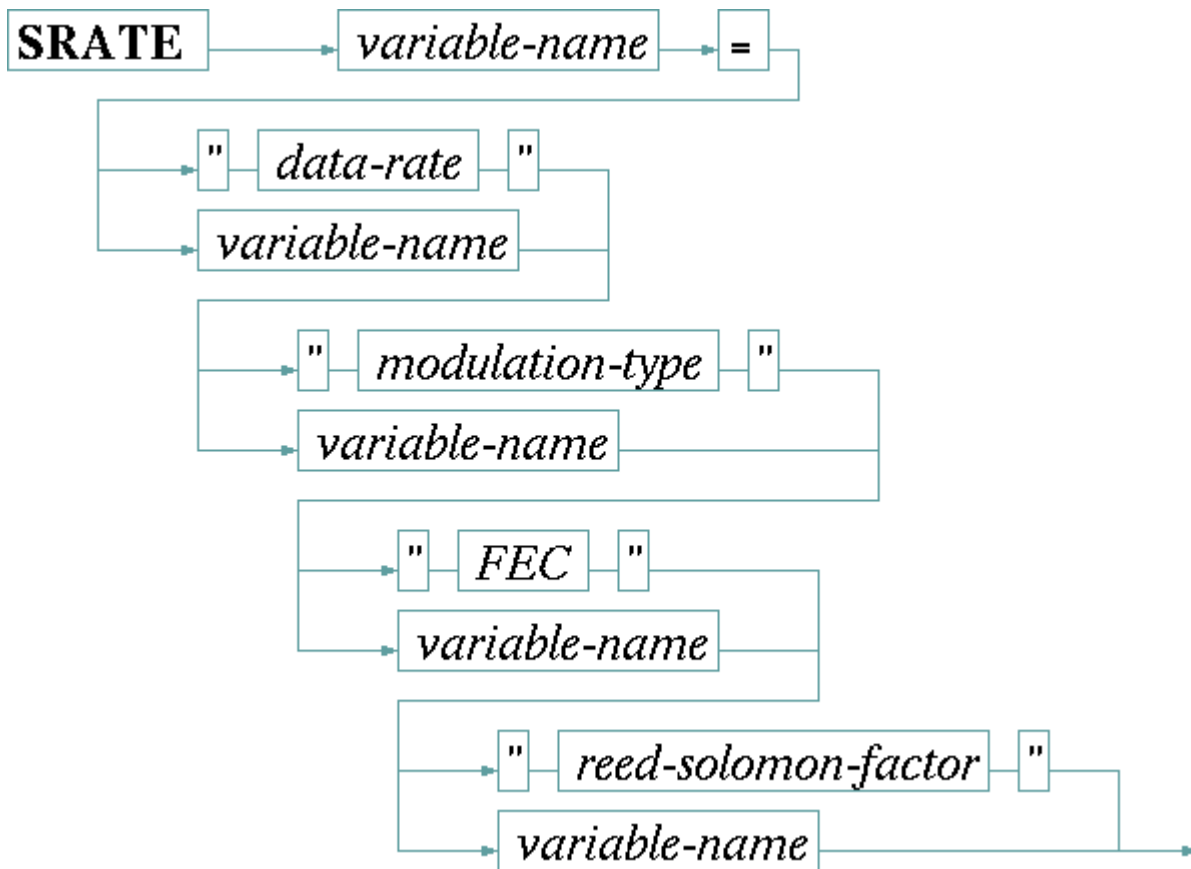
Reed-Solomon factor, computes an interface data rate from this and stores the result into the destination variable. The input data for the DRATE statement may be from constants given in double quotes or from the contents of driver variables.

The `DRATE` statement tries to guess the conversion factors from the input data following the rules described below:

- **modulation type**
  - Tries to guess the modulation type from the characters the token starts with. This catches all common names for modulation types.
  - `16*` : factor 4
  - `8*` : factor 3
  - `Q*` : factor 2
  - Everything else factor 1
- **FEC**
  - May be one of `1/2` , `2/3` , `3/4` , `4/5` , `5/6` , `6/7` , `7/8` or `8/9` .
  - Everything else is treated as `1/1` (no FEC).
- **Reed-Solomon factor**
  - Accepts any factor written as `nnn/mmm` , e.g. `188/204` .
  - If a single number is given, a denominator is assumed to be 204
  - Everything else is treated as `1/1` (no Reed-Solomon)

#### 1.2.10.4 The SRATE statement

The `SRATE` statement implements a data rate calculator which lets you convert a interface data rate into a symbol rate. This may be useful if a device driver shall supply both values, but the device itself only supports one of these settings.



The **SRATE** statement collects a data rate value, a modulation type, a FEC value and a Reed-Solomon factor, computes the symbol rate from this and stores the result into the destination variable. The input data for the **SRATE** statement may be from constants given in double quotes or from the contents of driver variables.

The **SRATE** statement tries to guess the conversion factors from the input data following the rules described below:

- **modulation type**

- Tries to guess the modulation type from the characters the token starts with. This catches all common names for modulation types.
- ``16*` : factor 4
- `8*` : factor 3
- `Q*` : factor 2
- Everything else : factor 1

- **FEC**

- May be one of `1/2` , `2/3` , `3/4` , `4/5` , `5/6` , `6/7` , `7/8` or `8/9` .
- Everything else is treated as `1/1` (no FEC).

- **Reed-Solomon factor**

- Accepts any factor written as `nnn/mmm` , e.g. `188/204` .
- If a single number is given, a denominator is assumed to be 204.
- Everything else is treated as `1/1` (no Reed-Solomon)

### 1.2.10.5 The WRITEHEX statement

The `WRITEHEX` statement interprets the contents of a variable as a hex dump of a binary byte array and sends this binary data to the device after packing it into a protocol frame.

**WRITEHEX** → *variable-name*

The `WRITEHEX` statement is used in the `StandardBin.nc` include file to define a convenient method of sending a command through the `lowLevel` driver variable for binary communication protocols. There is probably no other use for the `WRITEHEX` statement.

#### 1.2.10.6 The READHEX statement

The `READHEX` statement reads a message from the device, strips off the protocol frame and formats the received binary data as a hex dump. This is assigned to a driver variable.

**READHEX** → *variable-name*

The `READHEX` statement is used in the `StandardBin.nc` include file to define a convenient method of reading back the device's reply through the `lowLevel` driver variable for binary communication protocols. There is probably no other use for the `READHEX` statement.

#### 1.2.10.7 The INVALIDATE statement

The `INVALIDATE` statement marks a variable to be read from the device as soon as possible. It can be used if for instance a general mode setting has been changed at the device - requiring all parameter depending on this mode to be updated. `INVALIDATE` bypasses the *delayed read back* mechanism controlled by the `config.readBackDelay` setting, forces the read back in the same cycle if the `GET` procedure for the variable is located after the `INVALIDATE` statement within the driver.

**INVALIDATE** → *variable-name* →

For the `INVALIDATE` statement, use the keyword `INVALIDATE` followed by the name of the variable to be invalidated.

#### 1.2.10.8 The SYNC statement

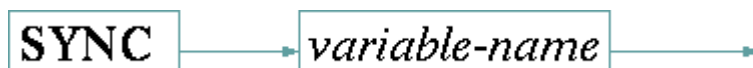
The `SYNC` statement *syncs* a variable, meaning that the variable's commanded value gets updated with the value recently read back from the device.

Normally, the device driver remembers a variable's value as it has been commanded from the GUI or from a logical device. When called in a `PRINT` or `WRITE` statement, this value will be used in favour of the value read back from the device. The driver uses this to check, if the commanded values has been accepted by the device.

When the device alters a commandable parameter autonomously, it may be desirable to accept

this value as the new value to be commanded. For example, imagine an antenna tracking controller device which accepts the pointing angles only as a triple az/el/pol. The operator commands all three angles to find the satellite and switches on tracking. The antenna controller tracks the satellite and as a consequence alters the pointing angles. If now one of the angles will be set in the GUI by the operator, the other two angles will revert to the values which had been set before tracking was switches on.

Using the SYNC statement after reading back the pointing values will help to avoid this situation.



For the `SYNC` statement, use the keyword `SYNC` followed by the name of the variable to be synced.

#### 1.2.10.9 The RAISECOMMFAULT statement

The `RAISECOMMFAULT` statement causes a communication fault to be raised. This e.g. can be used, if the device returns some value which is not expected or that indicates that the device did not recognize the last command. The fault gets cleared once the driver successfully passed one more cycle.



For the `RAISECOMMFAULT` statement, use the keyword `RAISECOMMFAULT` followed by a quoted string containing a message to be shown with the fault.

## 1.3 The RPN language extension

The **sat-nms** device driver definition language has been designed to be easy to understand and to provide a set of commands which is optimized to program device drivers. It enables people with some technical understanding but without any programming experience to add new device drivers to the **sat-nms** system.

Some complex devices require the device driver to behave more intelligent than the device driver definition language can do. For such cases the RPN language extension has been added to the device driver language.

The RPN language extension adds some functionality of an RPN (RPN = Reverse Polish Notation) programmable pocket calculator to the device driver. With RPN commands a device driver procedure can do:

- Arithmetic operations.
- String manipulation operations.
- Text based I/O to the device.
- Loops, conditional branches.

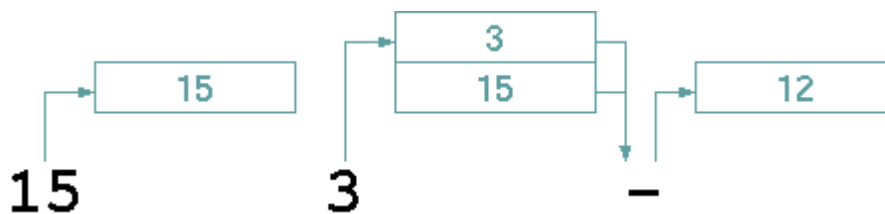


Device drivers using RPN commands are not as easy to understand as standard device drivers are. Only experienced programmers should use these commands as bad designed RPN command sequences may compromise the stability of the whole MNC system.

### 1.3.1 The RPN stack

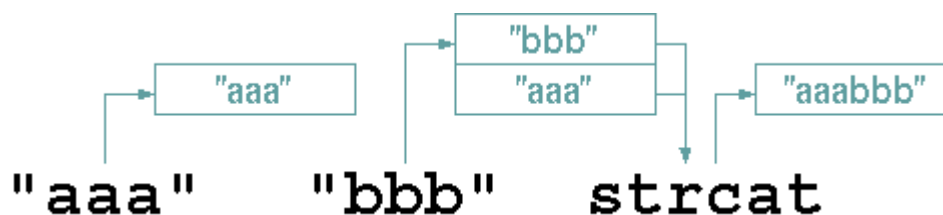
RPN (Reverse Polish Notation) is a way to define arithmetic operations which very efficiently can be used for computer programs. A number of pocket calculators use this method as well as programming languages like FORTH.

The example below illustrates how RPN works. Subtracting 3 from 15 you will be used to write as `15 3 -`. This is known as *infix* notation, the operator (`-`) is written in between the arguments it works on. With RPN (aka. *postfix* notation) the operator appears behind it's arguments:



- Writing `15` pushes this number on the stack.
- Now `3` pushes the second argument on the stack, the number `3` now appears on top of the `15`.
- Finally the `-` operator removes both numbers from the stack and replaces them by the result of the operation.

With the ***sat-nms*** device driver language, RPN operations are not limited to plain arithmetic. The example below illustrates a string concatenation:



The stack is able to store objects of different data types. Numbers, boolean values and character strings may be used. RPN functions automatically convert the type of their input data as required. The stack generally is not limited in size (the computer's memory size however limits the stack).

RPN operations may take an arbitrary number of parameters from the stack and they may leave an arbitrary number of results there. This makes functions possible which are much more complex than a simple `+` or `-`. Furthermore, the stack may be used as a storage for intermediate results. This together allows to perform quite complex operations very efficiently.

There is one stack for each device which keeps it's contents between driver procedures and

cycles. While this feature enables you to do very sophisticated things with the stack, you should be careful not to leave accidentally something on the stack at the end of an operation. The stack may grow with each cycle of the device driver, eating up slowly the whole available memory. Then, after hours, the MNC application aborts due to a lack of memory. There is a `clr` RPN command which clears the stack, you should call it at the end of an RPN sequence.

### 1.3.2 The { ... } statement

A sequence of words enclosed in curly braces is recognized by the device driver as an RPN command sequence. Within the RPN sequence commands must be separated by whitespace. Each word is interpreted as one of the following:

<b>Variable names</b>	cause the interpreter to push the variable's contents onto the stack.
<b>Table names</b>	cause the interpreter to remember this translation table and use it for the next <code>rxlt</code> / <code>xlt</code> command(s).
<b>Numeric (decimal) constants</b>	are pushed onto the stack as double precision floating point values.
<b>Character strings (in double quotes)</b>	are pushed as they are.
<b>Commands / keywords</b>	are executed as the <a href="#">RPN command reference</a> in the following chapter describes.

#### Example

```
{
  "ENQ;AUN" prt           // request the number of
  inp "=" find           // audio streams, get the reply
  !internal.numAudio      // and store it

  internal.numAudio if
  ""                      // audio stream list
  0                        // loop counter
  do
    "ENQ;AUX=" over "d04" fmt strcat prt // request one entry
    swap                      // get the list on top
    inp "=" find             // the complete entry
    dup "," trm 2 substr      // entry number 2 digits
    " " strcat               // delimiter
    swap "," find "," find   // language description
    strcat strcat "\n" strcat // append the reply
    swap                      // loop counter on top
    1 +                       // increment it
    internal.numAudio 1 -     // decrement numAudio,
    !internal.numAudio        // loop until there are more
    internal.numAudio while   // audio streams
    drop                      // the loop counter
  else
    "NONE\n"
  endif
  !audioList
  clr                        // due to paranoia ;-)
}
```

The example above - taken from the Tandberg-Alteia device driver - fills the variable `audioList` with a newline separated list of available audio streams as reported by the IRD. The RPN sequence first gets the number of available streams from the IRD and stores this into `internal.numAudio`. Then, unless `internal.numAudio` is zero, the program builds the list from the information returned by the IRD for each stream.

### 1.3.3 RPN command reference

The table below summarizes all commands and operations which are accepted by the device driver in RPN mode. The second column of the table describes how a command changes the stack. Left of the `--` the objects the operation takes from the stack are listed. The TOS (top of stack) is the rightmost value. Right of the `--` the results the operation leaves on the stack are shown.

command	stack usage	description
---------	-------------	-------------

command	stack usage	description
<i>variable-name</i>	-- v	Pushes the contents of the variable onto the stack. The type of the variable is retained ( CHOICE parameters are represented by string values).
<i>numeric-constant</i>	-- v	A numeric value (only decimal notation is allowed) is pushed as a floating point number onto the stack.
<i>%numeric-constant</i>	-- v	A numeric value (decimal) preceded by a % character is pushed as a long integer number onto the stack.
<i>#hex-constant</i>	-- v	A hexadecimal numeric value preceded by a # character is pushed as a long integer number onto the stack.
<i>"text"</i>	-- v	A text in double quotes is pushed onto the stack as a string constant.
<b>!</b> <i>variable-name</i>	v --	The exclamation mark, followed by a variable name, tells the driver to take one value from the stack and store it into the variable. Type conversion and range check is done as defined at the VAR statement which created the variable.
<b>^</b> <i>variable-name</i>	v --	The caret, followed by a variable name works much like store operator described above. However, while the store ( ! ) operator refers to the value the device driver read from the device, the caret operator loops the value to the driver as if an operator had commanded it.
<b>&lt;</b>	x y -- b	Compares x and y numerically. The result is true if $x < y$ .
<b>=</b>	x y -- b	Compares the string representation of x and y. The result is true if $x = y$ . Be careful when comparing floating point values.
<b>&gt;</b>	x y -- b	Compares x and y numerically. The result is true if $x > y$ .
<b>-</b>	x y -- z	Subtracts two values. The result is $x - y$ .
<b>/</b>	x y -- z	Divides two values. The result is $x / y$ . With $y = 0$ the result always is zero (although this is mathematically not correct).
<b>*</b>	x y -- z	Multiplies two values.
<b>+</b>	x y -- z	Adds two values.
<b>@</b>	's' -- z	Replaces the name of a parameter by its value. The parameter name must be a complete parameter ID, the value is that one displayed at the GUI, not the recently commanded one. If @ fails, the string 0 is left on the stack.

command	stack usage	description
<b>binand</b>	<b>x y -- z</b>	Performs a bitwise AND of two numeric operands.
<b>binneg</b>	<b>x -- z</b>	Binary negates the value on the stack.
<b>binor</b>	<b>x y -- z</b>	Performs a bitwise OR of two numeric operands.
<b>binxor</b>	<b>x y -- z</b>	Performs a bitwise XOR of two numeric operands.
<b>bit</b>	<b>n p -- b</b>	Isolates a bit value from a numeric value. Leaves a boolean flag on the stack which is true if the bit number <code>p</code> of the value <code>n</code> was set. Bit number 0 denotes the least significant bit in <code>n</code> .
<b>clr</b>	<b>... v --</b>	Clears the stack.
<b>copy</b>	<b>'s' 'd' --</b>	Copies a single parameter or a complete device setup from one device to another. <code>s</code> , <code>d</code> are interpreted as source and destination of the operation, must both be either plain device names or fully qualified parameter IDs (e.g. <code>MYDEVICE.grp.parName</code> ). When copying complete device setups, this is done via the device preset facility: a temporary / unnamed device preset is taken from the source device and applied to the destination device.
<b>cut</b>	<b>'s' n -- 's'</b>	Cuts the string <code>s</code> to the length of <code>n</code> characters. If <code>n</code> is greater than the length of the string to cut, the latter is left unchanged.
<b>debug</b>	<b>x --</b>	Takes a numeric value from the stack. Turns on debugging output if non-zero. The debugging output can be watched on the debug console window of the MNC software.
<b>do</b>	<b>--</b>	Starts a <b>do ... while</b> or a <b>do ... until</b> loop.
<b>drop</b>	<b>v --</b>	Removes one value from the stack.
<b>dup</b>	<b>v -- v v</b>	Duplicates the value on top of the stack.
<b>else</b>	<b>--</b>	Part of the <b>if ... else ... endif</b> construct.
<b>endif</b>	<b>--</b>	Closes an <b>if ... endif</b> or an <b>if ... else ... endif</b> construct.
<b>find</b>	<b>'s' 'p' -- 's'</b>	Finds the pattern <code>p</code> in the string <code>s</code> . Removes the beginning of <code>s</code> including the first occurrence of the pattern <code>p</code> . The result starts with the character following the first occurrence of <code>p</code> in <code>s</code> . If <code>p</code> is not found in <code>s</code> , find leaves an empty string on the stack.

command	stack usage	description
<b>fmt</b>	<b>n 'fmt'</b> <b>-- 's'</b>	Formats a numeric value according to format specification given in <code>fmt</code> . Leaves the formatted number as a string on the stack. The format specification follows the same rules the FMT option of the <a href="#">PRINT</a> statement does.
<b>func</b>	<b>v</b> <b>'fname'</b> <b>-- v</b>	Processes a numeric value through a tabular function using linear interpolation. First takes the name of the function to apply from the stack, then the value to process. The interpolated value is pushed back to the stack. Works much like the FUNCTION clause available in most I/O statements of the driver language. Chapter <a href="#">Function tables in I/O functions</a> gives an extensive description about functions and how they are used.
<b>hex</b>	<b>'x' -- n</b>	Interprets the top of the stack as the string representation of a hexadecimal number. Leaves the numeric value on the stack.
<b>if</b>	<b>n --</b>	Start a <b>if ... endif</b> or a <b>if ... else ... endif</b> clause. Takes one value from the stack. If this value is non-zero, the commands between <b>if</b> and <b>else</b> are executed.
<b>inp</b>	<b>-- 's'</b>	Reads one message from the device and places the data as a string on the stack. The protocol frame is removed.
<b>log10</b>	<b>x -- z</b>	This is the inverse function to <b>pow10</b> . It takes the common logarithm of the value on the TOS. To use this function to convert a <code>mW</code> expressed power to <code>dBm</code> follow this example: <code>mwval log10 10 *</code> converts the contents of <code>mwval</code> from <code>mW</code> to <code>dBm</code> .
<b>not</b>	<b>v -- v</b>	Negates the top of stack. Works with boolean values, numeric values (non-zero is turned into 0, 0 is turned into 1), and common string codings of boolean states.
<b>over</b>	<b>a b -- a</b> <b>b a</b>	Copies the value located one below the TOS on top of the stack.
<b>pow</b>	<b>x y -- z</b>	Raises <code>x</code> to the power of <code>y</code> . May be used to compute the square of a number ( <code>3 2 pow</code> leaves <code>9</code> on the stack). Floating point numbers are allowed for both operands, hence you may compute a square root like this: <code>myvar 0.5 pow</code> leaves the square root of the contents of <code>myvar</code> on the stack.
<b>pow10</b>	<b>x -- z</b>	Raises 10 to the power of <code>x</code> . This mainly is intended to convert RF power values expressed in <code>dBm</code> to <code>mW</code> . Example: <code>dbmval 10 / pow10</code> converts the contents of <code>dbmval</code> from <code>dBm</code> to <code>mW</code> , leaves the latter value on the stack.

command	stack usage	description
<b>prt</b>	's' --	Takes a value from the stack and sends its string representation to the device after adding the protocol frame.
<b>rot</b>	a b c -- b a c	Rotates three values on top of the stack.
<i>table</i> <b>rxlt</b>	's' -- 's'	Translates a string value through the given translation table. Translates from right to left. Uses the table with the name defined somewhere before the <b>rxlt</b> command. The table is <i>not</i> pushed to the stack, so the exact position of the table name does not matter.
<b>send</b>	'v' 'd' --	Sends a message to another device, setting a parameter at this device. <code>d</code> is interpreted as the full parameter ID of the message destination (e.g. <code>MYDEVICE.grp.parName</code> ), <code>v</code> is the value to be set there.
<b>shl</b>	'x' 'y' -- 'z'	Shifts <code>x</code> left by <code>y</code> positions. treats all values as long integers.
<b>shr</b>	'x' 'y' -- 'z'	Shifts <code>x</code> right by <code>y</code> positions. treats all values as long integers.
<b>strcat</b>	's' 's' -- 's'	Concatenates two string values.
<b>substr</b>	's' n -- 's'	Removes <code>n</code> character from the beginning of a string value.
<b>swap</b>	a b -- b a	Swaps two values on top of the stack.
<b>trm</b>	's' 'p' -- 's'	Cuts the string <code>s</code> at that point where the pattern <code>p</code> is found in the string. Leaves the string <code>s</code> unchanged if <code>p</code> does not exist in <code>s</code> .
<b>until</b>	n --	Closes a <b>do ... until</b> loop. Takes one value from the stack, loops again if this is zero.
<b>while</b>	--	Closes a <b>do ... while</b> loop. Takes one value from the stack, loops again if this is non-zero.
<i>table</i> <b>xlt</b>	's' -- 's'	Translates a string value through the given translation table. Translates from left to right. Uses the table with the name defined somewhere before the <b>xlt</b> command. The table is <i>not</i> pushed to the stack, so the exact position of the table name does not matter.

command	stack usage	description
---------	-------------	-------------

## 1.4 SNMP device classification and variable binding

As of version 3.1.610 the **sat-nms** MNC provides an universal SNMP agent which publishes a limited number of device parameters in a fixed, table oriented MIB. This SNMP agent permits to monitor and partially control any **sat-nms** MNC using the same MIB - regardless of the device configuration of the particular MNC.

The MIB provides one table which contains all devices which are configured in the **sat-nms** MNC. No special mapping is necessary for this table as there are only very common parameters like the device name, its fault state or its administrative state in this table.

Beside this, the MIB contains a couple of tables, each representing all devices belonging to a so called *device class*. For instance, there is one table containing all HPAs in the MNC and one table containing all waveguide switches. These class specific tables contain some parameters which are common to the devices of the particular class. The HPA table contains parameters for tx-on, for attenuation and for the measured output power for each device, the waveguide switch table provides a parameter for the switch position.

As device drivers sometimes use different parameter names for the same or similar function, an individual mapping from the *universal* parameter in the MIB to the real parameter in the device is necessary for every single **sat-nms** device driver. Device drivers which do not contain any information about the device class this device belongs to and how the parameters are to be mapped into the MIB only appear in the in the general device table of the MIB.

A **sat-nms** device (physical or logical) may be represented by multiple device classes in the MIB or by multiple instances of the same class. For example a combined upconverter and HPA may appear as a HPA in this table (with its HPA specific parameters) and a second time in the upconverter table with its frequency parameter.

Also devices consisting of multiple modules can be treated this way in the MIB, there are quad block upconverter on the market which provide four independent converter modules in one device. Such a device can be mapped in way, that it appears as four BUC devices in that table.

### 1.4.1 Device driver syntax for SNMP definitions

SNMP definitions are coded in the device driver file as special comment lines. This ensures, that new device drivers which include SNMP definitions can be run on older **sat-nms** systems which were delivered before this SNMP feature was introduced.



SNMP definitions involve two keywords, `@class` and `@varbind`. `@class` makes the device member of a certain device class and creates a row for this device in the table of the referenced device class. `@varbind` maps a SNMP OID to a certain **sat-nms** driver variable. All `@varbind` definitions following a `@class` definition refer to this and may only map OID names which are valid for the device class defined above.

The example below explains how to define the SNMP mapping for a device:

```
// @class HPA
// @varbind mncHpaTxOn tx.on
// @varbind mncHpaAttenuation tx.attn
// @varbind mncHpaOutputPower tx.measuredLevel
// @varbind mncHpaHelixCurrent meas.iHelix
// @varbind mncHpaState state
```

The `@class` definition identifies the device as a power amplifier and creates an instance for the device in the HPAs table. `@class` and the class name must appear in the same line and there must not be any trailing text right of the class name in the line. A list of valid class names may be found in chapter [List of SNMP device classes and OID names](#). The `@varbind` definitions below map the OIDs defined for one row of the HPA table to the corresponding variables in the driver. Each `@varbind` keyword must be followed by a valid OID name for this device class and then by the name of a device driver variable. Again, all three words must appear in the same line to be recognized correctly.

As mentioned above, a device may be part of multiple device classes at a time, it also may create multiple rows in the same SNMP table to map identical modules in the device to 'copies' of the device in the table. The example below illustrates this for a dual block upconverter:

```
// @class BUC 1
// @varbind mncBucTxOn tx.on.1
// @varbind mncBucInputAtten tx.attn.1
// @class BUC 2
// @varbind mncBucTxOn tx.on.2
// @varbind mncBucInputAtten tx.attn.2
```

Two `@class` definitions create two rows in the BUC table of the MIB. The number 1, 2 following the class name set the module index in the table, this can be used to differentiate between the two BUCs.

Device drivers maintained by SatService always list all `@varbind` OIDs defined for the `@class` of the device. Parameters which have no mapping for the particular device driver are listed with `!varbind` instead of `@varbind`. This prevents the parser from binding this OID to a variable of the driver, at the same time the complete list of `@varbind` OIDs stays visible in the device driver text for reference.

With the `mncTarSelTargets` `@varbind` (links the target names defined by a SatService ACU-ODM into the MIB), the software uses a `*` wildcard to denote the position of the target index in

the **sat-nms** variable name:

```
// @class TARSEL
// @varbind mncTarSelMoveToTarget    target.goto
// @varbind mncTarSelTargets         target.*
```

### 1.4.2 List of SNMP device classes and OID names

The following list shows the SNMP device classes known by the software and the valid OID names for each device class. Most OID names are self-explanatory. The MIB file `satnms-mnc-mib.txt` gives more detailed information about each value.

ACU	-- Antenna controller
mncAcuAzimuthState	r/o
mncAcuElevationState	r/o
mncAcuPolarizationState	r/o
mncAcuAzimuthTarget	r/w
mncAcuElevationTarget	r/w
mncAcuPolarizationTarget	r/w
mncAcuBeaconFrequency	r/w
mncAcuBeaconLevel	r/o
mncAcuMonopulseAzError	r/o
mncAcuMonopulseElError	r/o
mncAcuTrackingState	r/o
mncAcuTargetName	r/o
TRX	-- Tracking receiver
mncTrxFrequency	r/w
mncTrxBeaconLevel	r/o
mncTrxSignalOkState	r/o
mncTrxBeaconRelativeLevel	r/o
HPA	-- Power amplifier
mncHpaTxOn	r/w
mncHpaAttenuation	r/w
mncHpaOutputPower	r/o
mncHpaHelixCurrent	r/o
mncHpaState	r/o
mncHpaBeam	r/w
mncHpaOutputPowerWatt	r/o
mncHpaPrimaryTemp	r/o
mncHpaSecondaryTemp	r/o
mncHpaCurrent	r/o
mncHpaOperatingHours	r/o
EIRP	-- EIRP control device
mncEirpTxOn	r/w

mncEirpCommandedPower r/w  
mncEirpMeasuredPower r/o

BUC -- Block upconverter  
mncBucTxOn r/w  
mncBucFrequency r/w  
mncBucAttenuation r/w

UCONV -- Upconverter  
mncUconvTxOn r/w  
mncUconvFrequency r/w  
mncUconvAttenuation r/w

DCONV -- Downconverter  
mncDconvRxOn r/w  
mncDconvFrequency r/w  
mncDconvAttenuation r/w

TLT -- Test loop translator  
mncTltTxFrequency r/w  
mncTltRxFrequency r/w  
mncTltAttenuation r/w  
mncTltLoFrequency r/w

UPC -- Uplink power control  
mncUpcEnable r/w  
mncUpcInputDelta r/o  
mncUpcOutputDelta r/o

DEICE -- De-icing controller  
mncDeiceEnable r/w  
mncDeiceIceCondition r/o  
mncDeiceRainCondition r/o  
mncDeiceState r/o

DEHYD -- Waveguide pressurizer  
mncDehydEnable r/w  
mncDehydPressure r/o

WEATHER -- Weather monitor  
mncWeatherRain r/o  
mncWeatherWindSpeed r/o  
mncWeatherHumidity r/o  
mncWeatherPressure r/o  
mncWeatherTemperature r/o  
mncWeatherWindDirection r/o

WGS	- Waveguide switch
mncWgsPosition	r/w
mncWgsWrongPosition	r/o
mncWgsIndifferentPosition	r/o
SWITCH	-- On/off or 1:n switch
mncSwitchPosition	r/w
mncSwitchWrongPosition	r/o
mncSwitchIndifferentPosition	r/o
PS-11	-- 1:1 protection switch
mncPs11Protection	r/w
mncPs11ActiveChain	r/w
mncPs11Switched	r/o
PS-1N	-- 1:N protection switch
mncPs1nProtection	r/w
mncPs1nRedundantChain	r/w
mncPs1nSwitched	r/o
FOL	-- Fibre optic link
mncFolCardFaultState	r/o
mncFolCardType	r/o
mncFolCardAdminState	r/w
mncFolCardName	r/w
mncFolOptPower	r/o
mncFolRfPower	r/o
mncFolRequestedOptPower	r/w
mncFolRequestedRfPower	r/w
mncFolAgc	r/w
mncFolAttenuation	r/w
mncFolTemperature	r/o
mncFolLnbPower	r/w
mncFolLnbMeasCurrent	r/o
mncFolLnbMeasVoltage	r/o
PSU	-- Power supply unit
mncPsuUnitFaultState	r/o
mncPsuTemperature	r/o
mncPsuVoltage1	r/o
mncPsuVoltage2	r/o
mncPsuVoltage3	r/o
mncPsuVoltage4	r/o
TARSEL	-- Target selection
mncTarSelMoveToTarget	r/w
mncTarSelTargets	r/o

MOD	-- Modulator control
mncModInputStreamId	r/o
mncModInputStatus	r/o
mncModFrequency	r/w
mncModLevel	r/w
mncModTxOn	r/w
mncModSymbolRate	r/w
mncModStandard	r/w
mncModMode	r/w
mncModFec	r/w
mncModRollOff	r/w
mncModPilots	r/w

## 1.5 Device driver examples

The following pages contain two examples for a device driver, taken from the *sat-nms* device driver library. While the first driver is a quite simple implementation, the second driver makes use of the RPN language extension and other features.

- [NDSatCom-KuBand-Upconverter](#)
- [Tandberg-Alteia](#)

### 1.5.1 NDSatCom-KuBand-Upconverter Example

```
//
// Device driver for the ND SatCom Ku-band upconverter.
//
//
// CHANGE RECORD:
//
// 2001-05-05  1.00  initial version.
// 2001-06-23  1.01  the get procedure also depends on the tx.gain and
//                   the tx.frequency parameter.
// 2001-06-27  1.02  info.type set to NDSatCom-Upconverter
// 2001-08-09  1.03  changed info.type to match the file name.
//
COMMENT "NDSatCom-KuBand-Upconverter 1.03 010809"
PROTOCOL Miteq-MOD95
INCLUDE "drivers/Standard.inc"

/** identification variables *****/

VAR info.type    CYCLE 0  TEXT READONLY INIT "NDSatCom-KuBand-Upconverter"
VAR info.port    CYCLE 0  TEXT READONLY
VAR info.frame   CYCLE 0  TEXT READONLY INIT "Upconverter"
VAR info.if      CYCLE 0  TEXT READONLY
```

```

/** configuration variables *****/

/** MNC variables *****/

VAR tx.on          CHOICE "OFF|ON|"          CYCLE 2
VAR tx.gain        FLOAT 0 30.0 1 "dB"       CYCLE 0
VAR tx.frequency   FLOAT 12750.0 14500.0 3 "MHz" CYCLE 0

/** internal variables *****/

/** alarm flags *****/

ALARM faults.01    TEXT "Remote access"
ALARM faults.02    TEXT "Synthesizer"
ALARM faults.03    TEXT "LO-A lock"
ALARM faults.04    TEXT "LO-B lock"
ALARM faults.05    TEXT "Power supply"
ALARM faults.06    TEXT "IF-LO level"
ALARM faults.07    TEXT "RF-LO level"

/** overall status / parameter fetch routine *****/

TABLE T01 "OFF=1,ON=0"
TABLE T02 "70 MHz=0,140MHz=1"
TABLE T03 "0=1,1=0"
PROC GET WATCH tx.on tx.gain tx.frequency
  PRINT "A"
  INPUT AT 2 SCALE 0.001          tx.frequency
    AT 11 SCALE -0.1 OFFSET 30.0 tx.gain
    AT 15 CUT 1 XLT T03          faults.01
    AT 17 CUT 1 XLT T02          info.if
    AT 19 CUT 1 XLT T01          tx.on
    AT 21 CUT 1                  faults.02
    AT 22 CUT 1                  faults.03
    AT 23 CUT 1                  faults.04
    AT 24 CUT 1                  faults.05
    AT 25 CUT 1                  faults.06
    AT 26 CUT 1                  faults.07

/** set the tx.gain *****/

PROC PUT WATCH tx.gain
  PRINT "T" SCALE -10.0 OFFSET 300.0 FMT "d03" tx.gain
  INPUT
  DELAY 1.0

```

```

/** set the rf-on state *****/

TABLE T04 "OFF=M,ON=U"
PROC PUT WATCH tx.on
  PRINT XLT T04 tx.on
  INPUT

/** read / set the frequency *****/

PROC PUT WATCH tx.frequency
  PRINT "F" SCALE 1000.0 FMT "d8" tx.frequency
  INPUT
  DELAY 1.0

```

### 1.5.2 Tandberg-Alteia Example

```

//
// Device driver for the Tandberg Alteia and Alteia Plus DVB receivers.
//
// CHANGE HISTORY:
//
// 1.00 020505 created.
//

COMMENT "Tandberg-Alteia 1.00 020505"
PROTOCOL Tandberg-Alteia
INCLUDE "drivers/Standard.inc"

/** identification variables *****/

VAR info.type          CYCLE 0 TEXT READONLY INIT "Tandberg-Alteia"
VAR info.port          CYCLE 0 TEXT READONLY
VAR info.frame         CYCLE 0 TEXT READONLY INIT "IRD-Alteia"
VAR info.model         CYCLE 0 TEXT READONLY
VAR info.serial        CYCLE 0 TEXT READONLY
VAR info.ca.casid      CYCLE 0 TEXT READONLY
VAR info.ca.codeversion CYCLE 0 TEXT READONLY
VAR info.ca.bootversion CYCLE 0 TEXT READONLY
VAR info.ca.modelno    CYCLE 0 TEXT READONLY
VAR info.ca.hardware   CYCLE 0 TEXT READONLY
VAR info.ca.manufacturer CYCLE 0 TEXT READONLY
VAR info.ca.download   CYCLE 0 TEXT READONLY

/** configuration variables *****/

VAR config.lbandInputs  CYCLE 0 CHOICE "4,2" SETUP SAVE
VAR config.lbandPower  CYCLE 0 CHOICE "OFF,ON" SETUP SAVE

```

```
VAR config.inbPower      CYCLE 0 CHOICE "OFF,ON,BST" SETUP
VAR config.loFreq        CYCLE 0 FLOAT 0 0 1 "MHz" SETUP
VAR config.berThreshold  CYCLE 0 TEXT SETUP
VAR config.sigThreshold  CYCLE 0 INTEGER 0 255 "" SETUP
VAR config.errFrame      CYCLE 0 CHOICE "FREEZE,BLACK" SETUP
```

```
/** internal variables *****/
```

```
VAR internal.numServices CYCLE 0 INTEGER 0 0 "" READONLY
VAR internal.numAudio    CYCLE 0 INTEGER 0 0 "" READONLY
VAR internal.ack         CYCLE 0 TEXT          READONLY
VAR internal.flags       CYCLE 0 HEX    0 0 "" READONLY
VAR internal.scnt        CYCLE 0 INTEGER 0 0 "" READONLY INIT "0"
```

```
/** MNC variables *****/
```

```
VAR input          CYCLE 300 CHOICE "1,2,3,4"
VAR polarization    CYCLE 0  CHOICE "HOR,VER"
VAR frequency       CYCLE 0  FLOAT 0 0 1 "MHz"
VAR dataRate        CYCLE 0  FLOAT 0 0 3 "Mbps" READONLY
VAR symbolRate      CYCLE 300 FLOAT 0 0 3 "Msps"
VAR modulation      CYCLE 0  CHOICE "BPSK,QPSK,8PSK"
VAR fec             CYCLE 0  CHOICE "1/2,2/3,3/4,4/5,5/6,6/7,7/8,8/9"
VAR programNo       CYCLE 0  INTEGER 0 99 ""
```

```
VAR programList     CYCLE 2  TEXT READONLY
VAR audioList        CYCLE 3  TEXT READONLY
VAR actualProgram    CYCLE 2  TEXT READONLY
```

```
VAR video.fmt625     CYCLE 300 CHOICE "PALI,PALB,PALN"
VAR video.fmt525     CYCLE 0  CHOICE "NTSC,NTSN,PALM"
VAR video.level       CYCLE 0  INTEGER -30 30 "%"
VAR video.test        CYCLE 0  CHOICE "NRM,625.1,625.2,625.3,625.4"
                        "625.5,625.6,625.7,625.8,625.9"
                        "625.10,625.11,625.12,625.13"
                        "625.14,625.15,625.16,525.1"
                        "525.2,525.3,525.4,525.5,525.6"
                        "525.7,525.8,525.9,525.10,525.11"
                        "525.12,525.13,525.14,525.15,525.16"
```

```
VAR audio.1.program  CYCLE 0  INTEGER 0 0 ""
VAR audio.1.routing   CYCLE 0  CHOICE "NRM,MON,LFT,RGH"
VAR audio.1.output    CYCLE 0  CHOICE "ANALOG,AES/EBU,SPDIF,AC3"
VAR audio.1.level     CYCLE 0  INTEGER 6 18 "dB"
VAR audio.1.language  CYCLE 0  TEXT
VAR audio.1.test      CYCLE 0  CHOICE "NRM TEST_1 TEST_2 TEST_3 TEST_4 TEST_5"
```



```

VAR audio.1.test      CYCLE 0 CHOICE "NRM,TEST-1,TEST-2,TEST-3,TEST-4,TEST-5"
VAR audio.1.info      CYCLE 4  TEXT READONLY

VAR audio.2.program   CYCLE 0  INTEGER 0 0 ""
VAR audio.2.routing   CYCLE 0  CHOICE "NRM,MON,LFT,RGT"
VAR audio.2.output     CYCLE 0  CHOICE "ANALOG,AES/EBU,SPDIF,AC3"
VAR audio.2.level     CYCLE 0  INTEGER 6 18 "dB"
VAR audio.2.language  CYCLE 0  TEXT
VAR audio.2.test       CYCLE 0  CHOICE "NRM,TEST-1,TEST-2,TEST-3,TEST-4,TEST-5"
VAR audio.2.info       CYCLE 4  TEXT READONLY

VAR tsout.routing     CYCLE 300 CHOICE "PRE-CA,POST-CA"
VAR tsout.fibre       CYCLE 0  CHOICE "ON,OFF"

VAR flags.lock        CYCLE 1  BOOL READONLY
VAR flags.ca          CYCLE 0  BOOL READONLY
VAR flags.video       CYCLE 0  BOOL READONLY
VAR flags.audio       CYCLE 0  BOOL READONLY
VAR flags.ber         CYCLE 0  BOOL READONLY

VAR state.ber         CYCLE 0  TEXT          READONLY
VAR state.signal      CYCLE 0  INTEGER 0 255 "" READONLY
VAR state.aspect      CYCLE 3  TEXT          READONLY
VAR state.lines       CYCLE 3  TEXT          READONLY
VAR state.state       CYCLE 2  TEXT          READONLY

VAR ca.status         CYCLE 4  TEXT READONLY
VAR ca.service        CYCLE 0  TEXT READONLY
VAR ca.rasmode        CYCLE 0  CHOICE "DISABLED,FIXED,DSNG,SEC-CA"
VAR ca.bissmode       CYCLE 0  CHOICE "DISABLED,MODE 1,MODE E"
VAR ca.bisskey1       CYCLE 0  TEXT
VAR ca.bisskey2       CYCLE 0  TEXT
VAR ca.bissbits       CYCLE 0  TEXT
VAR ca.dsngkey        CYCLE 0  TEXT

VAR reset             CYCLE 0  TEXT

/** alarm flags *****/

ALARM faults.01 TEXT "Temperature"
ALARM faults.02 TEXT "Signal level"
ALARM faults.03 TEXT "Video lock"
ALARM faults.04 TEXT "Audio lock"
ALARM faults.05 TEXT "High BER"
ALARM faults.06 TEXT "Demodulator Lock"
ALARM faults.07 TEXT "Conditional Access"

```

```

/** translation tables *****/

TABLE tModulation  "BPSK=BPS,QPSK=QPS,8PSK=8PS"
TABLE tErrFrame    "FREEZE=FRZ,BLACK=BLK"
TABLE tVideoLevel  "-30=NEG30,-29=NEG29,-28=NEG28,-27=NEG27,-26=NEG26"
                  "-25=NEG25,-24=NEG24,-23=NEG23,-22=NEG22,-21=NEG21"
                  "-20=NEG20,-19=NEG19,-18=NEG18,-17=NEG17,-16=NEG16"
                  "-15=NEG15,-14=NEG14,-13=NEG13,-12=NEG12,-11=NEG11"
                  "-10=NEG10,-9=NEG09,-8=NEG08,-7=NEG07,-6=NEG06"
                  "-5=NEG05,-4=NEG04,-3=NEG03,-2=NEG02,-1=NEG01"
                  "0=POS00,1=POS01,2=POS02,3=POS03,4=POS04"
                  "5=POS05,6=POS06,7=POS07,8=POS08,9=POS09"
                  "10=POS10,11=POS11,12=POS12,13=POS13,14=POS14"
                  "15=POS15,16=POS16,17=POS17,18=POS18,19=POS19"
                  "20=POS20,21=POS21,22=POS22,23=POS23,24=POS24"
                  "25=POS25,26=POS26,27=POS27,28=POS28,29=POS29,30=POS30"
TABLE tVideoTest   "NRM=00,625.1=01,625.2=02,625.3=03,625.4=04,625.5=05"
                  "625.6=06,625.7=07,625.8=08,625.9=09,625.10=10,625.11=11"
                  "625.12=12,625.13=13,625.14=14,625.15=15,625.16=16"
                  "525.1=17,525.2=18,525.3=19,525.4=20,525.5=21,525.6=22"
                  "525.7=23,525.8=24,525.9=25,525.10=26,525.11=27,525.12=28"
                  "525.13=29,525.14=30,525.15=31,525.16=32"
TABLE tTsRouting   "PRE-CA=PRE,POST-CA=PST"
TABLE tTsFibre     "ON=ENA,OFF=DIS"
TABLE tAudioRouting "NRM=ST,MON=M1,LFT=M2,RGT=M3"
TABLE tAudioOutput  "ANALOG=ANA,AES/EBU=PRO,SPDIF=SPD,AC3=AC3"
TABLE tAudioTest    "NRM=0,TEST-1=1,TEST-2=2,TEST-3=3,TEST-4=4,TEST-5=5"
TABLE tRasMode      "DISABLED=DIS,FIXED=FIX,DSNG=DSN,SEC-CA=SCA"
TABLE tBissMode      "DISABLED=DIS,MODE 1=MO1,MODE E=MOE"
TABLE tCaStatus     "UNKNOWN CODE=FF"
                  "CARD INSERTED=00"
                  "CARD REMOVED=01"
                  "CARD INVALID=04"
                  "SERVICE BLOCKED=05"
                  "INVALID PACKET=06"
                  "CARD UNAUTHORIZED=07"
                  "HARDWARE FAILURE=08"
                  "CLEAR BUT RESTRICTED=09"
                  "SERVICE BLACKED OUT=10"
                  "SERVICE EXPIRED=11"
                  "CA WARNING=12"
                  "CA WARNING=13"
                  "PAIRING ERROR=14"
                  "CA WARNING=15"
                  "CA WARNING=16"
                  "CA WARNING=17"
                  "CA WARNING=21"

```

```
"CA WARNING=22"
TABLE tCaService "UNKNOWN CODE=XXX"
    "NO SERVICE SELECTED=NSS"
    "CLEAR=CLR"
    "RAS AUTHORIZED=RAS"
    "RAS UNAUTHORIZED=RAU"
    "BISS AUTHORIZED=BIA"
    "BISS UNAUTHORIZED=BIU"
    "VGUARD AUTHORIZED=GVA"
    "VGUARD UNAUTHORIZED=VGU"
    "NO CA INSTALLED=NCA"

/** procedures *****/

// called after sending a command to the IRD. checks the ACK response which is
// expected.
//
PROC SUBROUTINE getAck
    INPUT "=" TRM "_" internal.ack
    IF internal.ack = "ACK" GOTO endif
        SET faults.commstat = "ACK expected"
        SET faults.99 = "true"
    :endif

// called if the number of l-band inputs gets changed. sets the range for the
// "input" parameter
//
PROC PUT WATCH config.lbandInputs
    IF config.lbandInputs = "2" GOTO else
        RANGESET input CHOICES "1,2,3,4"
        GOTO endif
    :else
        RANGESET input CHOICES "1,2"
    :endif

// this procedure is executed once after the communication to the receiver has
// been established. it gets the device identification parameters and sets the
// LNB frequency according to the position of the input switch.
//
PROC GET WATCH info.serial info.model
    PRINT "REM;MOD"
    INPUT "MOD=" TRM "_" info.model
    PRINT "REM;SNM"
    INPUT "SNM=" TRM "_" info.serial
```

```
// reads the fault flags and derived the "LED flags" from these values
//
//
PROC GET WATCH flags.lock
  PRINT "OPR;SRQ"
  INPUT
    "SIG="      state.signal
    "ALR=" TRM " _ " internal.flags
    "BER=" TRM " _ " state.ber
  BITSET faults.01 = internal.flags 4 // Temperature
  BITSET faults.02 = internal.flags 5 // Signal level
  BITSET faults.03 = internal.flags 11 // Video lock
  BITSET faults.04 = internal.flags 9 // Audio lock
  BITSET faults.05 = internal.flags 13 // High BER
  BITSET faults.06 = internal.flags 15 // Demodulator lock
  BITSET faults.07 = internal.flags 14 // Conditional Access
  IF faults.06 = "true" SET flags.lock = "false"
  IF faults.07 = "true" SET flags.ca = "false"
  IF faults.03 = "true" SET flags.video = "false"
  IF faults.04 = "true" SET flags.audio = "false"
  IF faults.05 = "true" SET flags.ber = "false"
  IF faults.06 = "false" SET flags.lock = "true"
  IF faults.07 = "false" SET flags.ca = "true"
  IF faults.03 = "false" SET flags.video = "true"
  IF faults.04 = "false" SET flags.audio = "true"
  IF faults.05 = "false" SET flags.ber = "true"

// reads the service status message. this also is used to select a service
// after the IRD has decoded the signal
//
PROC GET WATCH state.state
{
  "SER;SRQ" prt inp
  "SRQ=" find ! state.state

  state.state "WAIT.PMT" = if
    internal.scnt 1 + !internal.scnt
    internal.scnt 6 > if
      "SER;SEL=00" prt inp
    else
      internal.scnt 3 > if
        "SER;SEL=" programNo "d02" fmt strcat prt inp
      endif
    endif
  else
    0 !internal.scnt
```

```
endif
}

// reads the information from the TUN;SRQ request.
//
PROC GET WATCH
    input frequency modulation polarization
    config.lnbPower config.loFreq
    PRINT "TUN;SRQ"
    INPUT
        "FRQ=" SCALE 0.125          frequency
        "LNB=" SCALE 0.125          config.loFreq
        "PWR=" CUT 3                 config.lnbPower
        "POL=" CUT 3                 polarization
        "RFI=" CUT 1                 input
        "MOD=" CUT 3 XLT tModulation modulation
    DRATE dataRate = symbolRate modulation fec "188"

// sets the RF input and the frequency. we switch both together to ensure that
// the input is set before the frequency is tuned.
//
PROC PUT WATCH input frequency
    PRINT "TUN;RFI=NO " input
    CALL getAck
    DELAY 3.0
    PRINT "TUN;FRQ=" SCALE 8.0 FMT "d06" frequency
    CALL getAck

// sets LO frequency (of the input actually set)
//
PROC PUT WATCH config.loFreq
    PRINT "TUN;LNB=" SCALE 8.0 FMT "d06" config.loFreq
    CALL getAck

// sets the modulation type
//
PROC PUT WATCH modulation
    PRINT "TUN;MOD=" XLT tModulation modulation
    CALL getAck

// sets the polarization by a 22kHz pulse
//
PROC PUT WATCH polarization
    PRINT "TUN;POL=" polarization
    CALL getAck

// reads the information from the DEM;SRQ request.
"
```

```
//
PROC GET WATCH fec symbolRate config.berThreshold config.sigThreshold
  PRINT "DEM;SRQ"
  INPUT
    "FEC=" CUT 3          fec
    "SYM=" TRM " _ " SCALE 0.0001 symbolRate
    "BER=" TRM " _ "      config.berThreshold
    "SIG=" CUT 3          config.sigThreshold
  DRATE dataRate = symbolRate modulation fec "188"

// sets the symbol rate
//
PROC PUT WATCH symbolRate
  PRINT "DEM;SYM=" SCALE 10000.0 FMT "d06" symbolRate
  CALL getAck

// sets the FEC
//
PROC PUT WATCH fec
  PRINT "DEM;FEC=" fec
  CALL getAck

// sets the BER threshold configuration parameter
//
PROC PUT WATCH config.berThreshold
  PRINT "DEM;BER=" config.berThreshold
  CALL getAck

// sets the signal level configuration parameter
//
PROC PUT WATCH config.sigThreshold
  PRINT "DEM;SIG=" FMT "d03" config.sigThreshold
  CALL getAck

// sets the LNB power configuration parameter
//
PROC PUT WATCH config.lnbPower
  PRINT "TUN;PWR=" config.lnbPower
  CALL getAck

//////////////////// VIDEO PARAMETERS //////////////////////

// gets the video state (and the video settings)
//
PROC GET WATCH video.fmt625 video.fmt525 video.level video.test
  state.aspect state.lines config.errFrame
  PRINT "VID;SRQ"
  INPUT
```

```
INFO 1
"625=" TRM " _ " video.fmt625
"525=" TRM " _ " video.fmt525
"ERR=" TRM " _ " XLT tErrFrame config.errFrame
"LVL=" TRM " _ " XLT tVideoLevel video.level
"OPS=" TRM " _ " XLT tVideoTest video.test
"VLS=" TRM " _ " state.lines
"SAR=" TRM " _ " state.aspect

// sets the 625 lines default video mode
//
PROC PUT WATCH video.fmt625
  PRINT "VID;625=" video.fmt625
  CALL getAck

// sets the 525 lines default video mode
//
PROC PUT WATCH video.fmt525
  PRINT "VID;525=" video.fmt525
  CALL getAck

// sets what to show is video is missing
//
PROC PUT WATCH config.errFrame
  PRINT "VID;ERR=" XLT tErrFrame config.errFrame
  CALL getAck

// sets the video level
//
PROC PUT WATCH video.level
  PRINT "VID;LVL=" XLT tVideoLevel video.level
  CALL getAck

// sets the video test mode
//
PROC PUT WATCH video.test
  PRINT "VID;OPS=" XLT tVideoTest video.test
  CALL getAck

////////// AUDIO CHANNEL 1 //////////

// gets the audio channel 1 parameters
//
PROC GET WATCH audio.1.routing audio.1.output audio.1.level audio.1.language
  audio.1.test audio.1.info
  PRINT "AUD;SRQ"
  INPUT
    "ROUT=" CUIT 2 XLT tAudioRouting audio.1.routing
```

```
PROC GET WATCH audio.1.routing audio.1.output audio.1.level audio.1.language audio.1.test audio.1.info
"LEV=M" CUT 2 audio.1.level
"OUT=" CUT 3 XLT tAudioOutput audio.1.output
"DFL=" CUT 3 audio.1.language
"OPS=" CUT 1 XLT tAudioTest audio.1.test
"CLN=" audio.1.info

// set the audio channel 1 output routing
//
PROC PUT WATCH audio.1.routing
PRINT "AUD;ROU=" XLT tAudioRouting audio.1.routing
CALL getAck

// set the audio channel 1 level
//
PROC PUT WATCH audio.1.level
PRINT "AUD;LEV=M" FMT "d02" audio.1.level
CALL getAck

// set the audio channel 1 output (hardware)
//
PROC PUT WATCH audio.1.output
PRINT "AUD;OUT=" XLT tAudioOutput audio.1.output
CALL getAck

// set the audio channel 1 default language
//
PROC PUT WATCH audio.1.language
PRINT "AUD;DFL=" audio.1.language
CALL getAck

// set the audio channel 1 test mode
//
PROC PUT WATCH audio.1.test
PRINT "AUD;OPS=" XLT tAudioTest audio.1.test
CALL getAck

//////////////////// AUDIO CHANNEL 2 //////////////////////

// gets the audio channel 2 parameters
//
PROC GET WATCH audio.2.routing audio.2.output audio.2.level audio.2.language
audio.2.test audio.2.info
PRINT "AU2;SRQ"
INPUT
"ROU=" CUT 2 XLT tAudioRouting audio.2.routing
"LEV=M" CUT 2 audio.2.level
```



```
"OUT=" CUT 3 XLT tAudioOutput audio.2.output
"DFL=" CUT 3 audio.2.language
"OPS=" CUT 1 XLT tAudioTest audio.2.test
"CLN=" audio.2.info

// set the audio channel 2 output routing
//
PROC PUT WATCH audio.2.routing
  PRINT "AU2;ROU=" XLT tAudioRouting audio.2.routing
  CALL getAck

// set the audio channel 2 level
//
PROC PUT WATCH audio.2.level
  PRINT "AU2;LEV=M" FMT "d02" audio.2.level
  CALL getAck

// set the audio channel 2 output (hardware)
//
PROC PUT WATCH audio.2.output
  PRINT "AU2;OUT=" XLT tAudioOutput audio.2.output
  CALL getAck

// set the audio channel 2 default language
//
PROC PUT WATCH audio.2.language
  PRINT "AU2;DFL=" audio.2.language
  CALL getAck

// set the audio channel 2 test mode
//
PROC PUT WATCH audio.2.test
  PRINT "AU2;OPS=" XLT tAudioTest audio.2.test
  CALL getAck

//////////////////// TRANSPORT STREAM OUTPUT //////////////////////

// reads the transport stream parameters
//
PROC GET WATCH tsout.routing tsout.fibre
  PRINT "TSO;SRQ"
  INPUT
    "CAM=" CUT 3 XLT tTsRouting tsout.routing
    "ENA=" CUT 3 XLT tTsFibre tsout.fibre

// sets the transport stream pre/post CA routing
```

```
//
PROC PUT WATCH tsout.routing
  PRINT "TSO;CAM=" XLT tTsRouting tsout.routing
  CALL getAck

// enables / disable the ASI fibre output
//
PROC PUT WATCH tsout.fibre
  PRINT "TSO;ENA=" XLT tTsFibre tsout.fibre
  CALL getAck

////////// CONDITIONAL ACCESS //////////

// reads the conditional access parameters. uses an RPN command sequence for
// this as - depending on the purchased configuration - the Alteia does not
// report all of these parameters all the time. RPN assigns empty strings to
// parameters which do not exist in the reply.
//
PROC GET WATCH ca.status ca.rasmode ca.bissmode
  ca.dsngkey ca.bisskey1 ca.bisskey2 ca.bissbits
{
  "CAS;SRQ" prt inp
  dup "CSS=" find " _ " trm tCaService rxlt ! ca.service
  dup "CST=" find " _ " trm tCaStatus rxlt ! ca.status
  dup "CID=" find " _ " trm ! info.ca.casid
  dup "CAC=" find " _ " trm ! info.ca.codeversion
  dup "BCV=" find " _ " trm ! info.ca.bootversion
  dup "CMN=" find " _ " trm ! info.ca.modelno
  dup "CHT=" find " _ " trm ! info.ca.hardware
  dup "CMA=" find " _ " trm ! info.ca.manufacturer
  dup "CDS=" find " _ " trm ! info.ca.download
  dup "RAM=" find " _ " trm ! tRasMode rxlt ca.rasmode
  dup "BIS=" find " _ " trm ! tBissMode rxlt ca.bissmode
  dup "BM1=" find " _ " trm ! ca.bisskey1
  dup "BM2=" find " _ " trm ! ca.bisskey2
  dup "BKE=" find " _ " trm ! ca.bissbits
  dup "DSK=" find " _ " trm ! ca.dsngkey
  clr
}

// sets the RAS mode parameter
//
PROC PUT WATCH ca.rasmode
  PRINT "CAS;RAM=" XLT tRasMode ca.rasmode
  CALL getAck
```

```
// sets the DSNG key used with one of the RAS modes
//
PROC PUT WATCH ca.dsngkey
  PRINT "CAS;DSK=" ca.dsngkey
  CALL getAck

// sets the BISS mode parameter
//
PROC PUT WATCH ca.bissmode
  PRINT "CAS;BIS=" XLT tBissMode ca.bissmode
  CALL getAck

// sets the BISS mode 1 key
//
PROC PUT WATCH ca.bisskey1
  PRINT "CAS;BM1=" ca.bisskey1
  CALL getAck

// sets the BISS mode 2 key
//
PROC PUT WATCH ca.bisskey2
  PRINT "CAS;BM2=" ca.bisskey2
  CALL getAck

// sets the BISS key length for modes 2/3
//
PROC PUT WATCH ca.bissbits
  PRINT "CAS;BKE=" ca.bissbits
  CALL getAck

////////// SERVICE SELECTION //////////

// reads the name of the actually selected service
//
PROC GET WATCH actualProgram
  PRINT "ENQ;NAM"
  INPUT "NAM=" actualProgram

// reads the service list (the bouquet) from the IRD and distributes the list
// as the variable programList. this proc is completely written in RPN language
// as this supports more flexible building the list from the particular
// entries.
//
PROC GET WATCH programList
{
```

```

state.state "WAIT.DEM" = if          // in WAIT.DEM state,
0 !internal.numServices              // no services are available
else
  "ENQ;NUM" prt                      // request the number of
  inp "=" find                      // services, get the reply
  !internal.numServices              // and store it
endif

internal.numServices if
  "ENQ;NET" prt inp "NET=" find      // network name
  "\n" strcat                       // as the first line of the list
  0                                  // loop counter
do
  "ENQ;SER=" over "d02" fmt strcat prt // request one entry
  swap                              // get the list on top
  inp "=" find dup                  // get the reply (2x)
  "," trm                           // the service index
  " " strcat                        // append " " to it
  swap "," find strcat              // append the service name
  strcat                            // append list entry
  "\n" strcat                       // trailing NL
  swap                              // loop counter on top
  1 +                                // increment it
  internal.numServices 1 -          // decrement numServices,
  !internal.numServices              // loop until there are more
  internal.numServices while        // services
  drop                              // the loop counter
else
  "NO PROGRAMS AVAILABLE\n"
endif
!programList
clr                                 // due to paranoia ;-)
}

// selects a service.
//
PROC PUT WATCH programNo
  PRINT "SER;SEL=" FMT "d02" programNo
  CALL getAck

// reads the audio list of the actually selected service from the IRD and
// distributes the list as the variable audioList. this proc is completely
// written in RPN language as this supports more flexible building the list
// from the particular entries.
//

```

```
//
PROC GET WATCH audioList
{
    "ENQ;AUN" prt                // request the number of
    inp "=" find                // audio streams, get the reply
    !internal.numAudio           // and store it

    internal.numAudio if
    ""                           // audio stream list
    0                             // loop counter
    do
        "ENQ;AUX=" over "d04" fmt strcat prt // request one entry
        swap                          // get the list on top
        inp "=" find                  // the complete entry
        dup "," trm 2 substr           // entry number 2 digits
        " " strcat                    // delimiter
        swap "," find "," find        // language description
        strcat strcat "\n" strcat     // append the reply
        swap                          // loop counter on top
        1 +                           // increment it
        internal.numAudio 1 -         // decrement numAudio,
        !internal.numAudio            // loop until there are more
        internal.numAudio while      // audio streams
        drop                          // the loop counter
    else
        "NONE\n"
    endif
    !audioList
    clr                               // due to paranoia ;-)
}

// selects the audio 1 stream
//
PROC PUT WATCH audio.1.program
    PRINT "SER;A1L=" FMT "d05" audio.1.program
    CALL getAck

// selects the audio 2 stream
//
PROC PUT WATCH audio.2.program
    PRINT "SER;A2L=" FMT "d05" audio.2.program
    CALL getAck
```

## 1.6 Device communication protocols

A device communication protocol encapsulates the recurrent operations to handle things like

start and stop characters or checksums with each message to be sent or received. The example below shows the complete message a ND-SatCom upconverter receives to tune its frequency. The pure command is `F1435000` (sets the frequency to 14,350.00 MHz). The protocol frame starts with a `{` followed by the device address. The command is terminated by `}` and finally a checksum character is sent.



A device communication protocol in the **sat-nms** software adds the protocol frame data (here marked red) to each message sent by a device driver with a `PRINT` or `WRITE` statement. The `PRINT` or `WRITE` statement creates the user data, the pure command. The protocol handler looks up the device address in the setup settings for the device, adds the start and stop characters and calculates the checksum over the message. The other way round, for each message received, the protocol handler strips off the protocol frame data and verifies the address and checksum fields. An `INPUT` or `READ` statement then receives the pure command data.

With the **sat-nms** software, new protocol definitions may be added to the software, simply by editing a text file for the new type of protocol frame which is needed.

### 1.6.1 Writing a communication protocol definition

As mentioned above, protocol definitions are coded as simple text files. When the MNC program starts, it *compiles* the protocols needed for its equipment setup to memory. Writing a protocol definition means editing such a text file and storing it at a place in the MNC computer where the MNC program searches for these files.

In a communication protocol definition file, you compose the protocol frame around a command from elements like characters, checksums or strings using a couple of keywords. The protocol definition language is quite simple, but powerful enough to specify almost all communication protocols used by satcom equipment.

A MNC system keeps all protocol definitions in a subdirectory called `protocols`. On standard installations this is the directory `/home/satnms/protocols`. The names of the protocol files consist of the protocol name followed by the extension `.proto`.

If you are writing the device driver on a Linux based MNC or NMS computer, you may want to use the VI text editor or `mcedit` (included in midnight commander) text editor for this. Both has been configured to colorize device driver files on these machines. We also providing syntax definitions for Visual Studio Code. Please ask your contact at SatService for details.

You also may copy protocol definition files to a MS-Windows based computer to edit them there. If you do this, you should consider the following:

- Protocol files are Unix based text files. Lines are terminated with a line-feed character only. Your favorite MS-Windows text editor may have problems to show these files.
- Unix / Linux is case sensitive with file names. Be sure that you don't mess up the case of characters in file name when you copy files between Unix and MS-Windows.

### 1.6.2 General file format

The protocol definition language uses a file format which is very similar to that one used with ***sat-nms*** device drivers. Like with device drivers the following applies:

- Whitespace (space characters, tabs, line breaks) separates words.
- Line breaks have no special termination function.
- All keywords are in upper case letters.
- Comments in C/C++ style are recognized (both, `/* ... */` and `// ...` comments).

Beside this, each protocol definition file has the same simple structure:

- The file starts with an (optional) comment block and some common definitions.
- Then, following the [TRANSMIT](#) keyword, the specification how a message shall be composed during transmit is given.
- Then the [RECEIVE](#) keyword starts the specification how to parse incoming data.

Below, an example for a protocol definition file is shown.

```
//
// The communication protocol used with SSE devices in NPI mode.
//

COMMENT "SSE-NPI 1.00 020108"
CLASS TTYProtocol

/** packet send procedure *****/

TRANSMIT

CHAR      2          // start byte, STX
CHAR      "F"        // source (master) LU
CHAR      "F"
ADDRESS    TEXT      // destination (slave) LU, 2 characters
HEXLENGTH  0          // 2 characters hex coded data length
USERDATA   // the command string
CHECKSUM   SUM8H 1 -1 // checksum 2 characters
CHAR      3          // end byte, ETX

/** packet receive procedure *****/

RECEIVE

START      2          // start byte, STX
ADDRESS    TEXT      // source (slave) LU
CHAR      "F"        // destination (master) LU
CHAR      "F"
HEXLENGTH  0          // 2 characters hex coded data length
USERDATA   // the response string
CHECKSUM   SUM8H 1 -1 2 // checksum 2 characters
CHAR      3          // end byte, ETX
```

### 1.6.3 Global definitions

The first section of the protocol definition file defines the protocol identification string and the Java class which is responsible to perform the I/O operations at runtime.

**COMMENT** → " *text* "

The **COMMENT** statement defines an identification string which is passed to the user interface. An operator can identify the type and version of a communication protocol used with a certain device.

The text following the **COMMENT** keyword is free field and principally may contain any



information. The device protocols coming with the **sat-nms** software all follow a convention which defines the comment string as

**"protocol-name X.YY YYMMDD"**

where **X** is the major version number of the protocol, **YY** the minor version number and **YYMMDD** the release date of this protocol version. It is recommended that customer defined device protocols follow this scheme, too.

**CLASS** → *class-name*

The **CLASS** statement defines the Java class used perform the protocol I/O steps during runtime of the MNC software. Actually **TTYProtocol** is the only protocol class which is applicable for customer defined protocol definitions.

Some communication protocols may be configured in some aspects of their functionality by applying protocol parameters to them. This is done with the **PROTOCOLPARAMETER** statement:

**PROTOCOLPARAMETER** → " *key=value* "

There may multiple **PROTOCOLPARAMETER** statements in one protocol definition file, each defining one aspect of the communication protocol. The **key** defines the parameter to set, **value** is the value to be assigned to this parameter.

Protocol parameters may be set either in the protocol definition file (thus defining the parameter for all uses of the protocol) or in the device driver. A definition in the device driver overwrites the setting in the protocol file.

**ATTENTION:** If you have multiple devices in the same device thread (**INTERFACE**) using different device drivers, the result is unpredictable if the drivers do not make exactly the same protocol parameter settings.

### Example

```
PROTOCOLPARAMETER "post.content.type=application/xml"
```

## 1.6.4 TX message elements

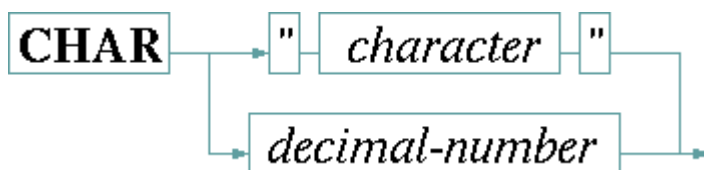
Starting with the **TRANSMIT** keyword, the second section of the protocol definition specifies the steps required to compose a valid protocol frame which is sent to the device.

The protocol frame definition consists of a sequence step specifiers, represented by a keyword followed by a defined number of parameters each. The step specifiers recognized in the the **TRANSMIT** section are:

<a href="#">CHAR</a>	Outputs a single character.
<a href="#">ADDRESS</a>	Outputs the device address.
<a href="#">USERDATA</a>	Outputs the user data.
<a href="#">CHECKSUM</a>	Outputs a message checksum.
<a href="#">DATALENGTH</a>	Outputs a data length field (binary).
<a href="#">HEXLENGTH</a>	Outputs a data length field (hex).
<a href="#">SEQUENCE</a>	Outputs a binary message sequence number.

#### 1.6.4.1 CHAR

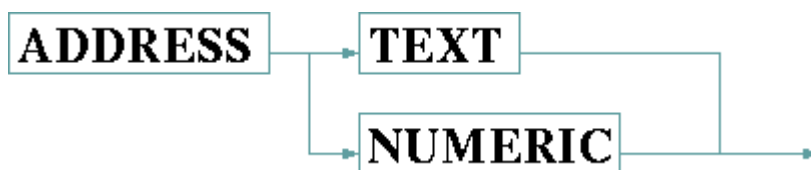
The CHAR step outputs a single character (byte).



The value may be specified as a single character in double quotes or as a decimal number in the range 0 ... 255 .

#### 1.6.4.2 ADDRESS

The ADDRESS step outputs the device address.



The ADDRESS keyword must be followed by one of TEXT or NUMERIC . If TEXT is specified, the address is output as a character string as it is entered at the user interface to the address configuration variable. If NUMERIC is specified, the address string is converted to a integer number and output as a single byte of this value.

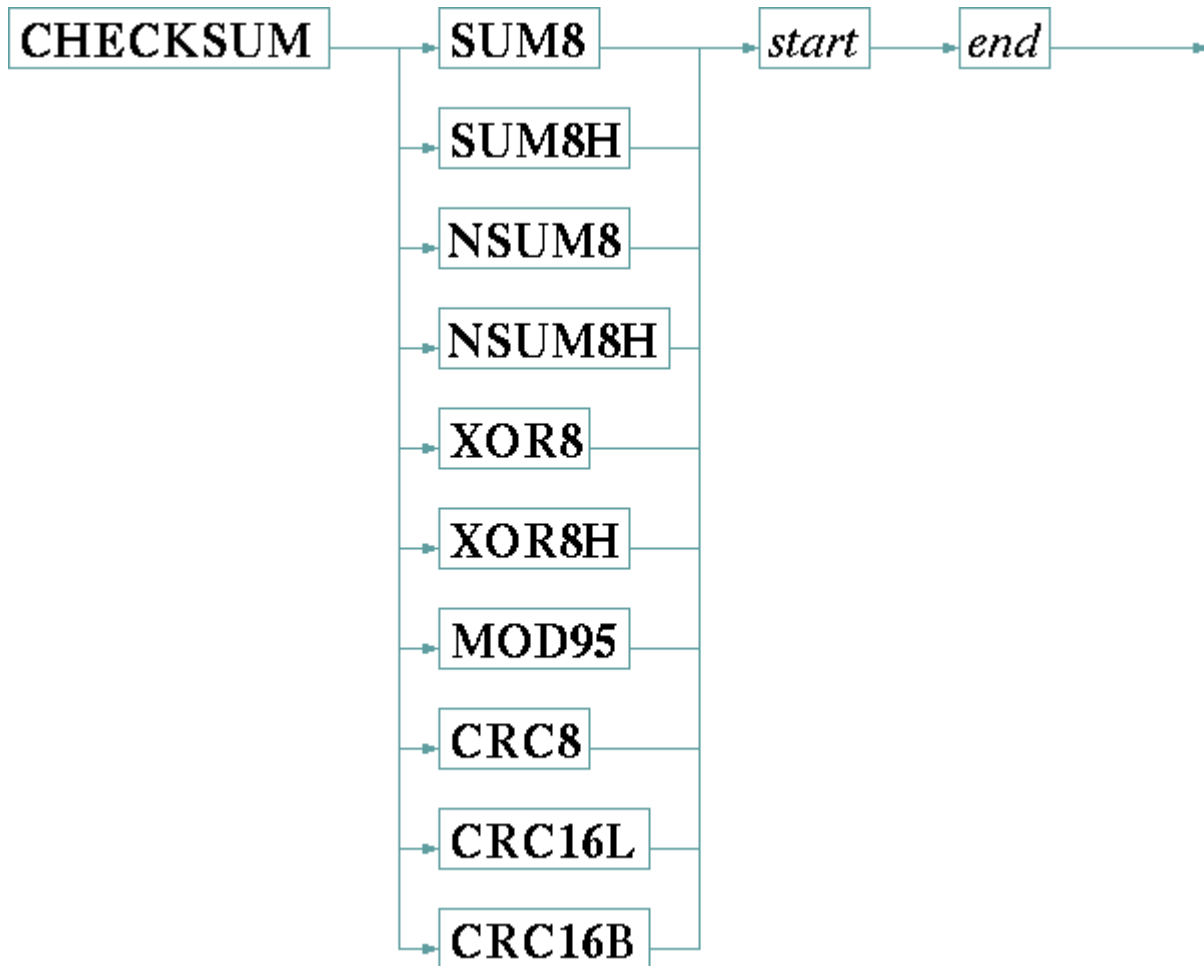
#### 1.6.4.3 USERDATA

The USERDATA step copies the user data as it was generated from the PRINT or WRITE statement in the device driver to the output.

**USERDATA**

#### 1.6.4.4 CHECKSUM

The `CHECKSUM` step outputs a checksum calculated from the message contents.



The parameters of this step specifier are:

<b>SUM8</b>	Creates a one byte checksum by adding all bytes and truncating the sum to the lowest 8 bits.
<b>SUM8H</b>	Like <code>SUM8</code> , but the checksum is coded as a 2 digit hex number.
<b>NSUM8</b>	Creates a one byte checksum by adding all bytes, taking the negative value of the sum and then truncating the sum to the lowest 8 bits.
<b>NSUM8H</b>	Like <code>NSUM8</code> , but the checksum is coded as a 2 digit hex number.
<b>XOR8</b>	Creates a one byte checksum by XOR-ing all specified bytes.
<b>XOR8H</b>	Like <code>XOR8</code> , but the checksum is coded as a 2 digit hex number.
<b>MOD95</b>	Creates the one character <i>modulo 95</i> checksum, known from the protocol used by Miteq devices (see below).

<b>CRC8</b>	Creates a one byte <b>CRC8</b> checksum.
<b>CRC16L</b>	Creates a two byte <b>CRC16</b> checksum, using the ARC/LHA CRC algorithm. Places the least significant byte first (little endian).
<b>CRC16B</b>	Creates a two byte <b>CRC16</b> checksum, using the ARC/LHA CRC algorithm. Places the most significant byte first (big endian).
<b>start</b>	The buffer position of the first character to be included in the checksum computation (0 means the first character).
<b>end</b>	The buffer position of the last character to be included in the checksum computation, relative to the actual position (-1 means the character left of the first checksum character).

The **MOD95** type checksum is computed following the formula:

$$Checksum = 32 + MOD_{95} \left[ \left( \sum_{i=1}^N message\ byte_i \right) - (32 * N) \right]$$

#### 1.6.4.5 DATALENGTH

The **DATALENGTH** protocol step outputs a packet length field.



The packet length is computed as the length of the user data string created by the **PRINT** or **WRITE** statement in the device driver plus the offset defined as a decimal number. It is coded as a one byte binary.

#### 1.6.4.6 HEXLENGTH

The **LENGTH** protocol step outputs a packet length field.



The length is computed as the length of the user data string created by the **PRINT** or **WRITE** statement in the device driver plus the offset defined as a decimal number. It is coded as a two character hex string.

#### 1.6.4.7 SEQUENCE

The **SEQUENCE** step outputs a one byte binary sequence number.



The sequence number gets incremented with each message that has been sent.

### 1.6.5 RX message elements

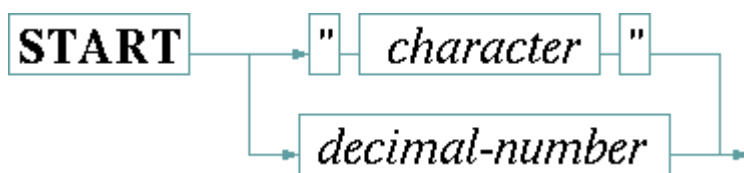
Starting with the `RECEIVE` keyword, the third section of the protocol definition specifies the steps required to parse a protocol frame which is received from the device.

The protocol frame definition consists of a sequence step specifiers, represented by a keyword followed by a defined number of parameters each. The step specifiers recognized in the the `RECEIVE` section are:

<a href="#"><u>START</u></a>	Reads a message start character.
<a href="#"><u>CHAR</u></a>	Reads a single character.
<a href="#"><u>ADDRESS</u></a>	Reads the device address.
<a href="#"><u>USERDATA</u></a>	Reads the user data.
<a href="#"><u>STRING</u></a>	Reads a terminated string as the user data
<a href="#"><u>CHECKSUM</u></a>	Reads a message checksum.
<a href="#"><u>DATALENGTH</u></a>	Reads a data length field (binary).
<a href="#"><u>HEXLENGTH</u></a>	Reads a data length field (hex).

#### 1.6.5.1 START

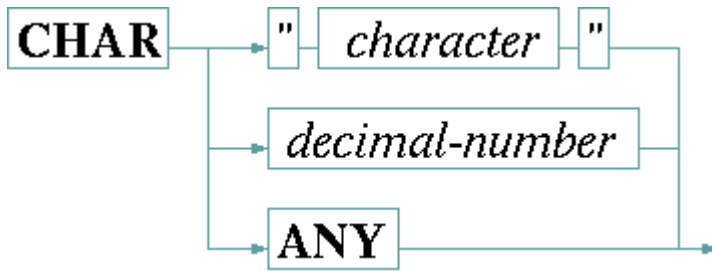
The `START` protocol step reads (and discards) all incoming data until the specified start character is received. The start character is not assumed to be part of the user data.



The character value may be specified as a single character in double quotes or as a decimal number in the range `0 ... 255`.

#### 1.6.5.2 CHAR

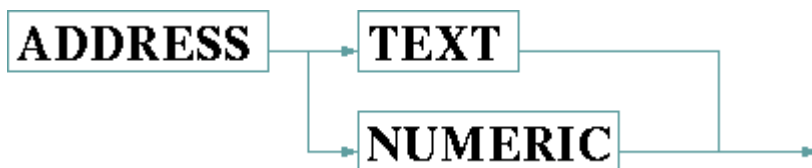
The `CHAR` step reads a single character and compares this to the expected value. If another character than expected has been received, the reception of this message is aborted and a communication fault is reported. The character received is not treated as part of the user data.



The value may be specified as a single character in double quotes or as a decimal number in the range 0 ... 255 . The special value ANY may be specified to make the CHAR step accept any character.

### 1.6.5.3 ADDRESS

The ADDRESS step reads the device address and compares it to the address value set in the device's setup menu. If the ADDRESS step received other data than expected, the reception of this message is aborted and a communication fault is reported.



The ADDRESS keyword must be followed by one of TEXT or NUMERIC :

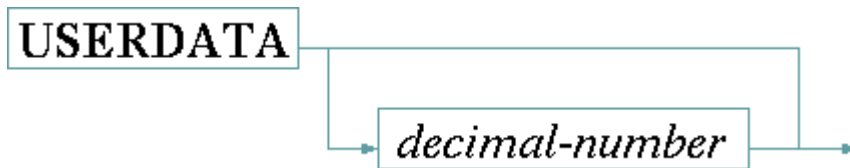
If TEXT is specified, the step reads as many characters as the address string configured for this device is long. The address is compared character by character to the received data.

If NUMERIC is specified, one character (byte) is read. The address string is converted to a integer number and compared to the received byte.

### 1.6.5.4 USERDATA

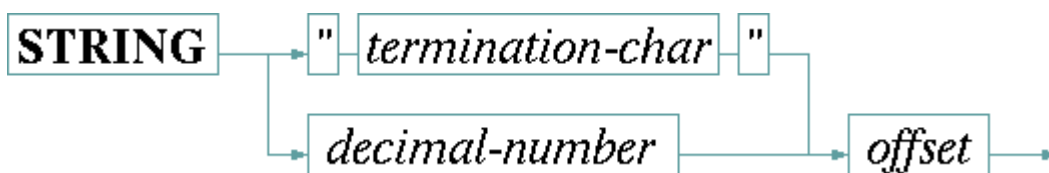
The USERDATA step reads a known number of bytes and returns these as the user data of the message. USERDATA requires to know the number of bytes to read, from one of these sources:

- A [DATALENGTH](#) protocol step appearing above the USERDATA step in the protocol definition.
- A [HEXLENGTH](#) protocol step appearing above the USERDATA step in the protocol definition.
- A fixed number of bytes, specified by a number following the USERDATA keyword. If the protocol definition contains a DATALENGTH/HEXLENGTH step and an explicitly given length in the USERDATA step as well, the length specified with the USERDATA step has precedence.



#### 1.6.5.5 STRING

The **STRING** step reads a character string of variable length which is terminated by a known character. The received data may be truncated by a certain number of character before it is returned as the user data part of the message.



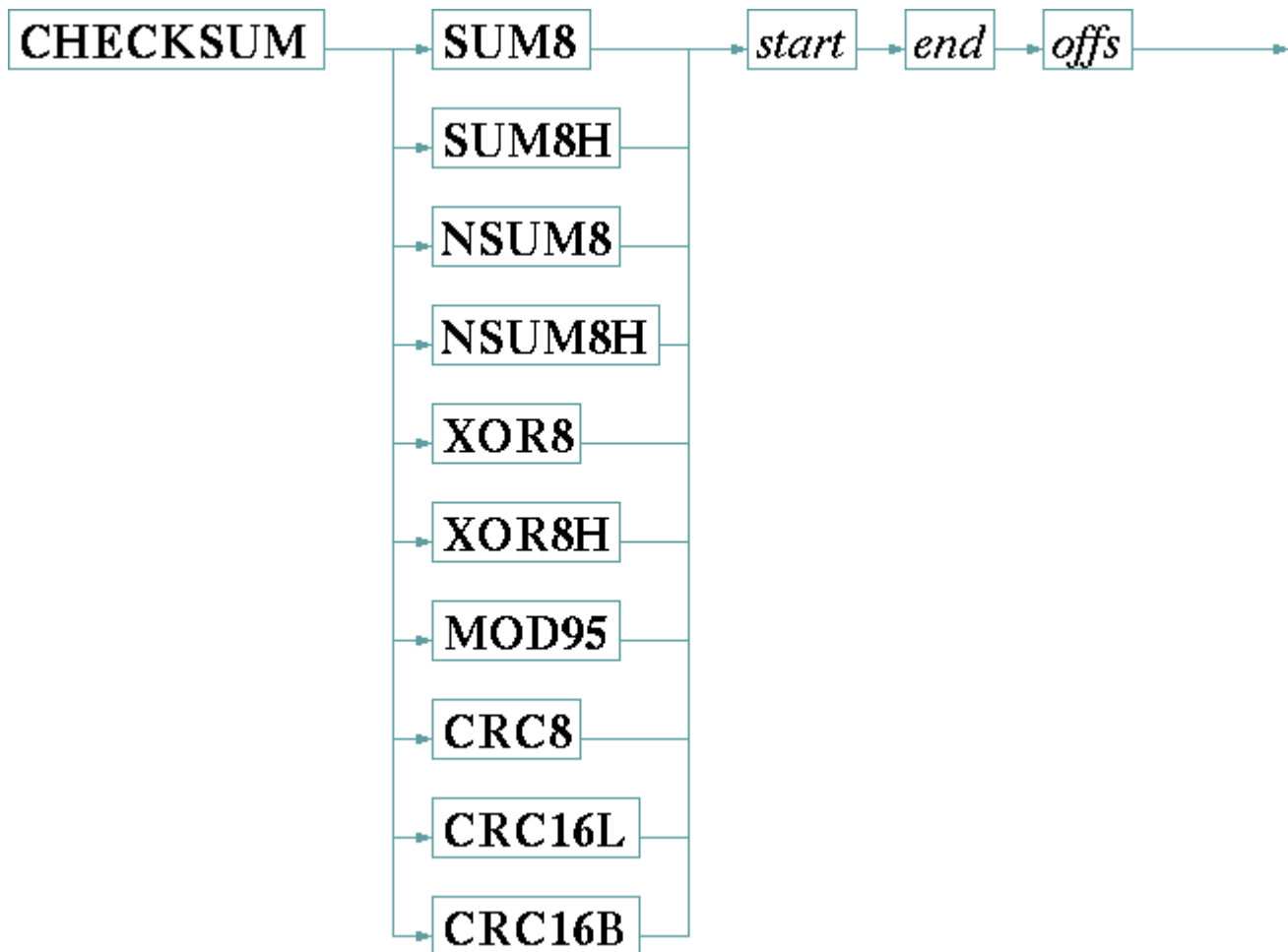
The terminating character may be specified either as a single character enclosed in double quoted or as a decimal number.

The **offset** parameter defines how many characters still have to be read or how many characters have to be cut at the end of the data to isolate the user data part of the message:

- If  $offset = 0$  the data read, including the termination character, is returned as the user data of the message.
- If  $offset < 0$  the given number of characters are cut off at the end of the data. E.g.  $offset = -1$  removes the termination character from the data.
- If  $offset > 0$  the given number of characters additionally is read and appended to the user data.

#### 1.6.5.6 CHECKSUM

The **CHECKSUM** step reads a checksum and compares it to the checksum calculated from the message contents. If the values differ, the reception of this message is aborted and a communication fault is reported.



The parameters of this step specifier are:

<b>SUM8</b>	Expects a one byte checksum done by adding all bytes and truncating the sum to the lowest 8 bits.
<b>SUM8H</b>	Like <code>SUM8</code> , but expects the checksum coded as a 2 digit hex number.
<b>NSUM8</b>	Expects a one byte checksum done by adding all bytes, taking the negative value of the sum and then truncating the sum to the lowest 8 bits.
<b>NSUM8H</b>	Like <code>NSUM8</code> , but expects the checksum coded as a 2 digit hex number.
<b>XOR8</b>	Expects a one byte checksum, done by XOR-ing all specified bytes.
<b>XOR8H</b>	Like <code>XOR8</code> , but expects the checksum coded as a 2 digit hex number.
<b>MOD95</b>	Expects the one character 'modulo 95' checksum, known from the protocol used by Miteq devices (see below).
<b>CRC8</b>	Expects a one byte <code>CRC8</code> checksum.



<b>CRC16L</b>	Expects a two byte <b>CRC16</b> checksum, calculated with the ARC/LHA CRC algorithm. Expects the least significant byte first (little endian).
<b>CRC16B</b>	Expects a two byte <b>CRC16</b> checksum, calculated with the ARC/LHA CRC algorithm. Expects the most significant byte first (big endian).
<b>start</b>	The buffer position of the first character to be included in the checksum computation (0 means the first character).
<b>end</b>	The buffer position of the last character to be included in the checksum computation, relative to the actual position ( <b>-1</b> means the last character read).
<b>offs</b>	The buffer position where the checksum starts, relative to the actual position ( <b>-1</b> means the last character read).

The **MOD95** type checksum is computed following the formula:

$$Checksum = 32 + MOD_{95} \left[ \left( \sum_{i=1}^N message\ byte_i \right) - (32 * N) \right]$$

#### 1.6.5.7 DATALENGTH

The **DATALENGTH** protocol step reads a one byte binary packet length field.



The received data length is remembered by the protocol until a [USERDATA](#) step is encountered. The [USERDATA](#) step will read this number of bytes *minus* the offset stated with the **DATALENGTH** step.

#### 1.6.5.8 HEXLENGTH

The **DATALENGTH** protocol step reads packet length field coded as a two byte hexadecimal number.



The received data length is remembered by the protocol until a [USERDATA](#) step is encountered. The [USERDATA](#) step will read this number of bytes *minus* the offset stated with the **HEXLENGTH** step.

## 1.7 Device oriented user interface

The **sat-nms** software provides a so called *device oriented* user interface for each type of equipment it supports. If you double click to a device icon at the MNC user interface you

automatically get a device window for this device. The software selects the right type of device window for the device driver which is configured for this device.

To use a device oriented user interface for the device drivers you added to the software by yourself, it is necessary to extend the framework which implements the device oriented user interface. The device oriented user interface for a particular device consists of one or more parameter screens which are made specially for this device type and a number of *universal* screens which appear with almost any device. So, there are principally two steps to do:

- Define the required parameter screens for the new device type
- Define the set of screens which shall be selectable in the device window.

### 1.7.1 How the software finds the screens for a device

To extend the device oriented user interface, it is important to understand how device drivers and the predefined user interface windows are linked together in the **sat-nms** software. Principally there are three steps from the device driver to the user interface:

1. Each device driver defines a variable called `info.frame`. The device driver sets this value to the name of the frame definition it expects.
2. The user interface software - when the operator clicks to a device icon - reads the frame definition file of this name in the `dframes` subdirectory which is usually located on standard installation in `/home/satnms/dframes`.
3. The frame definition file contains the list of screens (the names of the files defining the screen layouts). The user interface software builds a card folder like window from this information.

### 1.7.2 Creating new screens

The software keeps the files containing the screen layouts for the device oriented user interface in a directory called `dscreens` which is usually located on standard installation in `/home/satnms/dscreens`. There is one file for each parameter screen. The screens are created with the same tool used to design customer supplied screens for the task oriented user interface.

To protect the predefined screens from being accidentally modified, the layout editing tool normally has no access to these screens. To edit a screen for the device oriented user interface, run the client with the following command line options:

#### Linux / on the MNC server

```
java -cp client.jar satnms3.guiconf.Configurator localhost /dscreens/...
```

where `...` is the name of the screen to edit. This opens the layout editing tool with the given file name. Please note, that the MNC server must be running at this time. The layout editing tool reads the file via the data base capabilities of the server rather than accessing the disk file directly.

**Linux / on a remote machine** You also may start the layout editing tool on a client computer over the LAN. At this place open a terminal session and enter:

```
java -cp client.jar satnms3.guiconf.Configurator [ip] /dscreens/...
```

Like above, ... has to be replaced by the name of the screen to edit. [ip] stands for the IP address of the MNC server.

**Windows / on a remote machine** You also may start the layout editing tool on a MS Windows based client computer over the LAN. At this place open a command prompt window or create a shortcut with the following command line:

```
java -cp client.jar satnms3.guiconf.Configurator [ip] /dscreens/...**
```

Like above, ... has to be replaced by the name of the screen to edit. [ip] stands for the IP address of the MNC server. If your computer does not recognize the `java` command, try **jre** instead and check the installation of your Java Runtime Environment.

Editing the screen layout for the parameter screens is straight forward. It is recommended to use a layout similar to the existing screens. The software's user manual describes the usage of the layout editing program and introduces the types of display elements a screen may consist of.

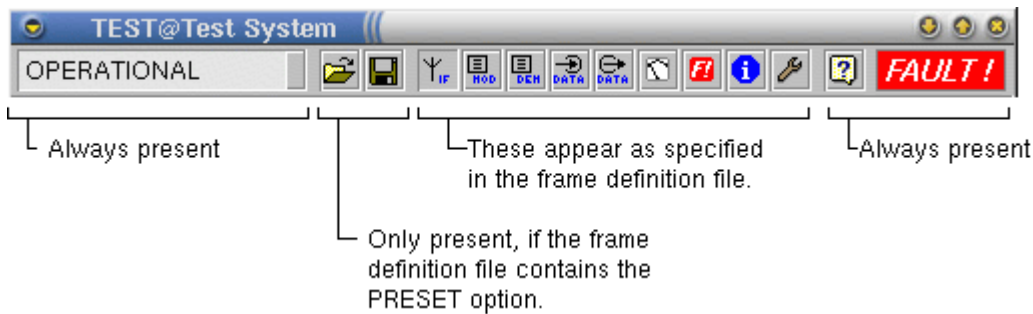
### 1.7.3 Creating a 'frame' definition

The set of screens making up the window for a particular device type is called a frame. Frames are defined by text files located in the `~/dframes` directory of the software installation. To add a new frame type to the software, you have to edit such a file.

#### PRESETS

tb-if.gif;	/dscreens/Modem-General;	General / IF parameters
tb-mod.gif;	/dscreens/Modem-TX;	Modulator parameters
tb-dem.gif;	/dscreens/Modem-RX;	Demodulator parameters
tb-dinput.gif;	/dscreens/Modem-TX-IFC;	TX interface parameters
tb-doutput.gif;	/dscreens/Modem-RX-IFC;	RX interface parameters
tb-meter.gif;	/dscreens/Modem-Measure;	Meter Readings
tb-fault.gif;	@FAULTS;	Faults and fault mask
tb-info.gif;	@INFO;	Device Info
tb-tool.gif;	@CONFIG;	Maintenance

The example above shows a frame definition file for a SCPC satellite modem. Lines starting with a '#' are comments, they are ignored when the file is read. Below the button bar of the device window resulting from this frame definition is shown:



- The operation mode selector on the left and the help button and fault mark at the right are contained in all device screens, they need not to be specified in the frame definition file.
- The **PRESET** keyword in the third line of the frame definition file tells the software to display the buttons for parameter preset storage and retrieval in the button bar of the device window. The preset load / save buttons by default require the operator to be logged in with privilege level 100 or higher. You may append a number to the **PRESET** keyword, separated by a blank character in order to specify the minimum privilege level for load / save preset buttons in this device window explicitly. **PRESET 150** for example makes the buttons requires privilege level 150 or higher.

The table starting at line five of the file specifies the other buttons in the tool bar and the screens bound to them. The buttons appear in the same order as the lines of the table. Each table row consists of three fields, separated by semicolon characters:

- The **first** entry selects the icon to be displayed. The following chapter shows the icons available with their file names.
- The **second** entry specifies the screen that shall be shown on a click to this button. This either is the name of a screen file (usually from the **dcreens** directory) or one of these keywords selecting a special screen:
  - **@FAULTS** : Shows the common screen listing the faults of this device.
  - **@INFO** : Shows the common screen listing the info variables for this device.
  - **@CONFIG** : Shows the common maintenance and configuration screen for this device.
  - **@SATLIST** : Shows a screen interfacing to an antenna pointing database (used by the AntennaPointing logical device)
- The **third** field defines the the help text for this button which is shown above the mouse cursor if this id hold above the button.

















































All device frames provided by SatService contain the @FAULTS, @INFO and @CONFIG screens in the way shown in the example.

### 1.7.4 Icon reference

column 1	column 2	column 3	column 4
----------	----------	----------	----------

column 1	column 2	column 3	column 4
1 tb-1.gif	<sup>12</sup> <sub>3</sub> tb-123.gif	2 tb-2.gif	3 tb-3.gif
4 tb-4.gif	5 tb-5.gif	6 tb-6.gif	<sup>123</sup> <sub>ABC</sub> tb-a1.gif
<sup>abc</sup> tb-abc.gif	✓ tb-accept.gif	✉ tb-ackmail.gif	 tb-ainput.gif
 tb-aoutput.gif	 tb-apsdvb.gif	 tb-apsskywan.gif	 tb-asiout.gif
 tb-async.gif	<sup>1</sup>  tb-audio1.gif	<sup>2</sup>  tb-audio2.gif	<sup>3</sup>  tb-audio3.gif
<sup>4</sup>  tb-audio4.gif	<sup>5</sup>  tb-audio5.gif	<sup>6</sup>  tb-audio6.gif	<sup>7</sup>  tb-audio7.gif
<sup>8</sup>  tb-audio8.gif	 tb-aupc.gif	 tb-avoutput.gif	AZ tb-az.gif
 tb-back.gif	 tb-backup.gif	 tb-c2n.gif	 tb-cam.gif
 tb-cgroup.gif	 tb-chain.gif	 tb-chanlist.gif	 tb-chart.gif
 tb-chdir.gif	✓ tb-checkmark.gif	 tb-chhist.gif	 tb-clock.gif
 tb-colors.gif	<sup>AB</sup> tb-compare.gif	 tb-condacc.gif	 tb-connect.gif
 tb-connflt.gif	 tb-copy.gif	 tb-cross.gif	 tb-cut.gif
 tb-data.gif	 tb-decode.gif	 tb-delete.gif	 tb-dem.gif
 tb-device.gif	 tb-devices.gif	 tb-devicex.gif	 tb-dflash.gif
 tb-dial.gif	 tb-dialing.gif	 tb-diallist.gif	 tb-dinout.gif
 tb-dinput.gif	 tb-dirup.gif	 tb-disconnect.gif	 tb-doutput.gif
▼ tb-downarrow.gif	 tb-draw.gif	→ tb-dst.gif	 tb-dvbs2.gif
EL tb-el.gif	 tb-encode.gif	 tb-erasor.gif	! tb-exclam.gif
 tb-exec.gif	 tb-eye.gif	⌘ <sub>0</sub> tb-farray0.gif	⌘ <sub>1</sub> tb-farray1.gif
⌘ <sub>2</sub> tb-farray2.gif	⌘ <sub>3</sub> tb-farray3.gif	⌘ <sub>4</sub> tb-farray4.gif	⌘ <sub>5</sub> tb-farray5.gif
⌘ <sub>6</sub> tb-farray6.gif	⌘ <sub>7</sub> tb-farray7.gif	⌘ <sub>8</sub> tb-farray8.gif	▼ tb-fastdown.gif
▶▶ tb-fastfwd.gif	◀ tb-fastrew.gif	▶ tb-fastup.gif	 tb-fault.gif
 tb-faultlist.gif	 tb-faultlog.gif	 tb-filexfer.gif	 tb-frame.gif

column 1	column 2	column 3	column 4
 tb-frm.gif	 tb-ftrack.gif	 tb-geopos.gif	 tb-gflash.gif
 tb-go.gif	 tb-gref.gif	 tb-grid.gif	 tb-groups.gif
 tb-hangup.gif	 tb-help.gif	 tb-ibs.gif	 tb-idr.gif
 tb-if.gif	 tb-image.gif	 tb-info.gif	 tb-input.gif
 tb-ipinout.gif	 tb-leftarrow.gif	 tb-line.gif	 tb-link.gif
 tb-linklist.gif	 tb-list.gif	 tb-livebelt.gif	 tb-livelog.gif
 tb-lnb.gif	 tb-lock.gif	 tb-locked.gif	 tb-login.gif
 tb-magnify.gif	 tb-mail.gif	 tb-master.gif	 tb-measlist.gif
 tb-meter.gif	 tb-mod.gif	 tb-modems.gif	 tb-move.gif
 tb-mpegout.gif	 tb-mux.gif	 tb-new.gif	 tb-newchild.gif
 tb-newdev.gif	 tb-newdevs.gif	 tb-newifc.gif	 tb-newitem.gif
 tb-newnode.gif	 tb-nosound.gif	 tb-now.gif	 tb-nowarn.gif
 tb-onestep.gif	 tb-open.gif	 tb-output.gif	 tb-paste.gif
 tb-pause.gif	 tb-play.gif	 tb-plug.gif	 tb-pmirror.gif
 tb-pmlchk.gif	 tb-pmrchk.gif	 tb-preset.gif	 tb-print.gif
 tb-properties.gif	 tb-question.gif	 tb-queue.gif	 tb-record.gif
 tb-rectangle.gif	 tb-red.gif	 tb-redial.gif	 tb-redstop.gif
 tb-relay.gif	 tb-reload.gif	 tb-restore.gif	 tb-rf.gif
 tb-rflash.gif	 tb-rightarrow.gif	 tb-rmail.gif	 tb-route.gif
 tb-rs.gif	 tb-rx.gif	 tb-sainteract.gif	 tb-sat-a.gif
 tb-sat-c.gif	 tb-sat-m.gif	 tb-sat-opt.gif	 tb-sat-p.gif
 tb-sat-s.gif	 tb-sat-st.gif	 tb-sat1.gif	 tb-sat2.gif
 tb-sat3.gif	 tb-sat4.gif	 tb-sat5.gif	 tb-satellite.gif
 tb-satout.gif	 tb-save.gif	 tb-saveas.gif	 tb-savevlc.gif
 tb-schedule.gif	 tb-search.gif	 tb-send.gif	 tb-setdev.gif

column 1	column 2	column 3	column 4
 tb-sigma.gif	 tb-slave.gif	 tb-slavelink.gif	 tb-smodem.gif
 tb-sound.gif	 tb-spectrum.gif	 tb-spider.gif	 tb-src.gif
 tb-srcdst.gif	 tb-stop.gif	 tb-switch.gif	 tb-swoff.gif
 tb-swon.gif	 tb-text.gif	 tb-textend.gif	 tb-tmove.gif
 tb-tool.gif	 tb-tools.gif	 tb-transmitting.gif	 tb-trash.gif
 tb-tree.gif	 tb-treeedit.gif	 tb-tsanalyze.gif	 tb-tsoutput.gif
 tb-tx.gif	 tb-undo.gif	 tb-unlocked.gif	 tb-uparrow.gif
 tb-vgroup.gif	 tb-vinput.gif	 tb-vlc.gif	 tb-vlclist.gif
 tb-vlcmmap.gif	 tb-voptim.gif	 tb-voutput.gif	 tb-vsearch.gif
 tb-vupdate.gif	 tb-wizard.gif	 tb-xzoomfull.gif	 tb-xzoomin.gif
 tb-xzoomout.gif	 tb-yflash.gif	 tb-ystar.png	 tb-yzoomfull.gif
 tb-yzoomin.gif	 tb-yzoomout.gif	 tb-zoomin.gif	 tb-zoomout.gif

## 1.8 Online Help

The **sat-nms** software provides a context sensitive online help based on HTML files. To make this help system easy to maintain, a simple type setting language has been defined for it. The source of the help system is contained in a couple of Markdown files containing plain text and some formatting commands. A help compiler creates all the HTML files for the online help from the sources.

With the **sat-nms** software there come beside a ready compiled online help also the help compiler and all sources of the help system. This enables you to extend the online help system for your needs.

You however should be aware, that the original help files (sources) are restored with each software update you install at the system. To avoid conflicts with the documentation supplied by SatService, you should modify or extend the online help only in the following situation:

1. You have written your own device driver, and you want to supply a help page for this type of equipment.

### 1.8.1 Help file format

Help files are now simply defined in Markdown a lightweight markup language for creating formatted text using a plain-text editor and are stored in `.md` files.

We are supporting the standard Markdown as described at <https://commonmark.org/>.

If you want to have your own help files included into the online help system, please provide the new driver definition and your help files to SatService GmbH and we will automatically build the new online help with your additional files.

**Attention!** The old ***sat-nms*** help file format was replaced by this standardized Markdown files.

For your reference please find below this deprecated information:

So the following chapter describes the format of the source files the online manual of the software is built from.

## Topic Definitions

The online manual divides the complete text into topics, where each topic is compiles to one web page. A topic is identified by it's unique four digit topic number. This number is used to build the file name of the web page and also to reference the page in hyper links.

In the source text a topic starts with a topic definition line and ends where the next topic starts. The topic definition looks like this:

**.topic TTTT L The title of this topic**

The keyword '.topic' must start at the very first column of the line. Further on, TTTT is the for topic number of the topic to define. L is a number in the range 1..4 defining the level of this heading in the table of contents. The remaining part of the line defines the title of the topic.

## Paragraphs

The HTML text is shown by the browser as left justified text, made up to the width of the browser window. Line breaks and paragraphs do **not** appear where you type them in the source code, you have explicitly to insert line break or paragraph break commands into the source text to get these breaks in the output.

<b>.p</b>	Closes a paragraph. Most browsers show this as a line break followed by a half height blank line.
<b>.br</b>	Inserts a line break.

Both commands may appear anywhere in the line, however, usually they are placed either at the line end or in the own line.

## Formatted Lists

You may typeset ordered (numbered) lists and unordered (bullet) lists in the text. Browsers usually display each list item as a paragraph, preceded by either the paragraph number or a



list item symbol. You may nest lists of both types to any depth. The help compiler simply translates the list formatting commands to HTML, it does not check if each list is properly terminated. If you produce invalid HTML code because of missing list end marks, some browsers will give strange results.

<b>.ul</b>	Starts a bullet list.
<b>.ol</b>	Starts an ordered (numbered) list
<b>.li</b>	Starts a list item.
<b>.br</b>	Inserts a line break.
<b>.eul</b>	Closes a bullet list.
<b>.eol</b>	Closes an ordered (numbered) list

Although these commands are recognized everywhere in the line, formatting lists as shown in the example below makes the source well readable.

```
.ol
.li This is the first list item
.li This is the second one, it contains a lot of text which does not fit in one line.
.li The third item is a short one again.
.eol
```

## Tables

Help pages may contain tables consisting of an almost arbitrary number of columns. Tables appear with a pale blue background. You have no control over the precise layout of the table, it is shown as the browser likes to display plain HTML tables with a cell background set. Because no geometry must be specified, it is very simple to define the table in the source text:

<b>.tl</b>	Starts a table definition
<b>.ts</b>	Starts the first cell of a table row
<b>.tc</b>	Separates a table cell from the next one in the same row
<b>.te</b>	Marks the end of the last table cell in a row.
<b>.etl</b>	Closes the table definition.

With the commands shown above, the table gets defined cell by cell, row by row. Table definitions are translated by the help compiler one by one to HTML tags. When defining a table, be careful to define the same number of cells in each row and to terminate each row properly by a **.te** command. The example below shows in which way a simple table definition should be

formatted in the source:

```
.tl
.ts cell 1/1 .tc cell 1/2 .tc cell 1/3 .te
.ts cell 2/1 .tc cell 2/2 .tc cell 2/3 .te
.ts cell 3/1 .tc cell 3/2 .tc cell 3/3 .te
.etl
```

## Images / Pictures

The user manual also contains images and diagrams. Files in GIF or JPG format may be included. The image files must be located in the same directory as the other files of the online help. You find already a bunch of image files there, most of them are copies of the tool bar icons used by the software.

<b>.i file-name</b>	Includes an image.
---------------------	--------------------

If this is the only command in the line, be sure to add a single space character after the file name. This is due to a flaw of the help compiler which does not recognize the file name properly if it ends at the line end.

## Pre-formatted Text

The text may include segments of 'pre-formatted' text, which the browser displays using a mono-spaced font with line breaks at the same positions as in the source. Pre-formatted text is used to include for example excerpts of configuration files.

<b>.pre</b>	Starts pre-formatted text
<b>.epre</b>	Ends pre-formatted text

The `.pre` and `.epre` commands must appear as single commands in a line, starting at the first column.

## Emphasizing Text

Portions of text may be *emphasized*, **bold printed** or set in a `typewriter font`. To do this, enclose the text to highlight in curly braces and insert a letter `e` for emphasized, `b` for bold or `t` for typewrite directly after the opening brace. There must no be a space character between the opening brace and the code character. The opening and closing braces must be in the same line, nesting different hi-lighting styles is not allowed.

## Cross-references / Hyper-links

You may include HTML hyper-links into the text to permit the reader to jump directly from one topic to another by a single mouse click. To insert a hyper-link, To do this, enclose the text to

hi-light in braces and insert the sequence `h####` directly after the opening brace. `####` is to be replaced by the topic number to link to. Like with the character formatting codes above, there must no be a space character between the opening brace and the `h`. The opening and closing braces must be in the same line, no nesting of braces is allowed.

## Paragraph Headings

There is no special command to insert paragraph headings. The topic title is the only heading for a help page. Only this will appear in the table of contents in the printed version.

To introduce headings on a paragraph level, use a bold printed, one line paragraph instead.

## Escaping Dot Commands

Words in the help text containing dots (e.g. file names) may be confused with a dot command sequence by the help compiler. Example: `MyDevice.proto` in the source file will be printed as `MyDevice` at the end of one paragraph and `roto` at the beginning of the next one. This is because the `.p` sequence in the file name is translated to a paragraph break.

The `.dot` sequence is used to escape the dot command in such a situation. Spelling the file name `MyDevice.proto` in the source produces the desired output.

## Comments

You may add comments or annotations to the source text of the document which later are not shown in the manual. Lines which start with a double percent character `%%` at the first column are skipped when the file is processed by the help compiler.

## Conditional Compilation

The help compiler implements a mechanism to include or skip parts of the source text depending on a couple of environment information here called *features*. The conditional compilation mechanism closely follows that one used with common programming language compilers:

<b>%if feature</b>	Marks the start of the documents part which only shall be included if 'feature' is present with this software installation
<b>%else</b>	Marks the end of <code>%if</code> branch, starts another one with the negated condition. The <code>%else</code> statement is optional
<b>%endif</b>	Marks the end of this <code>%if</code> or <code>%if</code> / <code>%else</code> clause.

The `%if`, `%else` and `%endif` statements must appear singly in a line and must start at the first column in the line.

The following 'features' may be tested in an `%if` clause (testing other features, always results in *not present*):

### 1.8.2 Rebuilding the online help

The `help` directory of the **sat-nms** software installation contains a shell script called `makehelp`. If you change to the `help` directory and call this script there, the complete online help system gets rebuilt. You should do this in one of the following cases:

- You have added a device driver to the software.
- You have edited the help file for a new device driver.
- You have edited the 'local.hlp' file.
- You installed a software update.

### 1.8.3 Adding help files for new devices

To display a context sensitive help for each device driver, the help system uses a separate file for each device type, containing some information how to use and to configure the driver/device. This help topic is called if you click to the help button in a device's window.

Device driver help files use a slightly different format than the main help file does. The difference is, the a device driver help file must not contain topic definitions. One topic definition for each file is generated automatically when the help system is built.

Device driver help files are located in the `help` directory of the **sat-nms** software installation. The files are named like the drivers itself, but with the extension `.hlp`. The help compiler creates dummy files for missing device driver files.

The help files for the device drivers supplied by SatService all follow a common format. They first explain for which type of equipment this driver was designed for. The the button bar of the device window is described in a table. Below this the configuration/setup parameters for this device driver are explained, some device specific remarks may follow. You are encouraged to use this scheme for your customer supplied device drivers. too.

## 1.9 ServiceClient library

The **sat-nms** ServiceClient application lets you easily define virtual devices and group them to *services* which show a subset of important parameters of these devices and a summarized fault state of all devices of a service.

The ServiceClient library defines the device types the ServiceClient knows and the implementation of these devices types by **sat-nms** device drivers.

The ServiceClient library is an XML file which is read by the ServiceClient application at startup. You may modify this file to make the ServiceClient recognize additional **sat-nms** device drivers as implementation of a predefined device type or you even may define additional device types from scratch and add some implementations to them. Beside the device type definitions the XML file contains some settings which influence the GUI appearance of the ServiceClient.

The following chapters describe how to add new [device types and implementations](#) to the library and how to [adapt GUI settings](#) like the colors used for signalling the fault state of devices and services. Later chapters then describe the [file format of the XML file](#) in detail.

### 1.9.1 Adding new devices or device types to the library

To make the **sat-nms** ServiceClient recognize devices of a new type, there must be at least one `implementation` for the device type defined in the

- Devices containing multiple instances
- Fault detection
- Conditional implementation

### 1.9.2 Adapting the appearance of the ServiceClient

Some aspects of the ServiceClient's GUI appearance are defined in a section of the `definitions.xml` file. These are beside others the colors used to signal states like `OK`, `WARNING` or `FAULT` in the GUI. You may change these settings by editing the `definitions.xml` file, with the next start of the ServiceClient the changed settings will be used.

Chapter [GUI constants](#) later in this document gives a complete list of GUI constants which may be adapted by editing this file.

### 1.9.3 File format of definitions.xml

### 1.9.4 Device type definitions

#### 1.9.4.1 Parameters

#### 1.9.4.2 Implementations

### 1.9.5 GUI constants

The `definitions.xml` file contains a section `<gui ... </gui>` which defines a number for constant definitions which let you adapt some aspects of the program's appearance to your needs. Every definition in this section has the form:

```
<def name="..." value="..." />
```

For each definition, `name` is a unique name for the constant, `value` is the value assigned to it. All definitions have default values which apply if the definition is not found.

Below, a table shows all known constant definitions with a description of their meaning and the default values the software assumes if the definition is not found in the `definitions.xml` file.

name	type	default	description
------	------	---------	-------------

name	type	default	description
<b>editorPrivilegeLevel</b>	integer	150	The minimum privilege level an operator must have in order to edit the service configuration. For operators below this privilege level the toolbar button for the edit mode is without function.
<b>leftPaneWidth</b>	integer	250	The width (pixels) of the "service" screen elements which make up the left screen pane. Usually there is no need to change this except are dealing with very long service names or comment strings which do not fit in the default width.
<b>rightPaneWidth</b>	integer	600	The width (pixels) of the screen elements in the right screen pane which show the parameters of a device in the service. The width of these screen elements does not automatically grow with the number of parameters shown in it, so it may be necessary to enlarge this if you are going to define new device types with many parameter in one row. The default width is sufficient to display three "wide" and one "narrow" parameters in a single row.
<b>okBackgroundColor</b>	color	00FF00 (green)	The background color used to signal that an element is in OK state. The service widgets on the left pane, the header of the service detailed view on the right pane and the fault indicator of each device are using this color definition. The color is expressed as a 6-digit hexadecimal RGB value defining the red, green and blue component of the color.

name	type	default	description
<b>warningBackgroundColor</b>	color	FFFF00 (yellow)	The background color used to signal that an element is in WARNING state. The service widgets on the left pane, the header of the service detailed view on the right pane and the fault indicator of each device are using this color definition. The color is expressed as a 6-digit hexadecimal RGB value.
<b>faultBackgroundColor</b>	color	FF0000 (red)	The background color used to signal that an element is in FAULT state. The service widgets on the left pane, the header of the service detailed view on the right pane and the fault indicator of each device are using this color definition. The color is expressed as a 6-digit hexadecimal RGB value.
<b>okForegroundColor</b>	color	000000 (black)	The text / foreground color used to signal that an element is in OK state. If you change the "okBackgroundColor", you may want to adapt the color used to the foreground as well to maintain sufficient contrast
<b>warningForegroundColor</b>	color	000000 (black)	The text / foreground color used to signal that an element is in WARNING state. If you change the "okBackgroundColor", you may want to adapt the color used to the foreground as well to maintain sufficient contrast
<b>faultForegroundColor</b>	color	FFFFFF (white)	The text / foreground color used to signal that an element is in FAULT state. If you change the "okBackgroundColor", you may want to adapt the color used to the foreground as well to maintain sufficient contrast

If you want to revert a value to its default and do not know the default setting, we recommend

to change the 'def' tag into something other (e.g. usedefault) instead of deleting the line defining this definition. This way the name of the definition is still in the file, but due to the modified tag it will not be read.

## 1.10 Reference Tables

This chapter provides a number of tables with reference information which may be useful as a quick lookup for questions regarding the extension of the *sat-nms* software. In particular, the supplied tables are:

- [Device driver keyword reference](#)
- [Protocol definition keyword reference](#)
- [Help file keyword reference](#)
- [Debugging with terminal session](#)
- [Global faults](#)

### 1.10.1 Device driver keyword reference

The following table lists the keywords recognized in a device driver specification file in alphabetical order. When viewing this table online you may use the hyper-links in the description column to jump to the place where this keyword is explained in the manual.

<b>=</b>	Part of the syntax of a <a href="#">SET</a> or <a href="#">BITSET</a> statement.
<b>ALARM</b>	The <a href="#">ALARM</a> statement defines a faults flag.
<b>AT</b>	Within an <a href="#">INPUT</a> statement, the AT clause moves the read cursor to a certain position in the received data.
<b>BIGENDIAN</b>	Specifies that a <a href="#">WRITE</a> or <a href="#">READ</a> statement shall treat multibyte integer numbers in big endian (MSB first) byte order.
<b>BIT</b>	Reads a single bit from an integer number within an <a href="#">INPUT</a> statement.
<b>BITMERGE</b>	Should no longer be used.
<b>BITS</b>	Specifies to write or read a number of bits with in a certain byte in a <a href="#">WRITE</a> or a <a href="#">READ</a> statement.
<b>BITSET</b>	The <a href="#">BITSET</a> statement isolates a single bit from a numeric variable and assigns it to another one.
<b>BITSPLIT</b>	Should no longer be used.
<b>BOOL</b>	Defines a variable to be a boolean flag in a <a href="#">VAR</a> statement.
<b>CALL</b>	The <a href="#">CALL</a> statement calls a procedure as a subroutine.



<b>CBER</b>	Interprets a two character string as a bit error rate within an <a href="#">INPUT</a> statement.
<b>CHOICE</b>	Defines a variable to be a choice list in a <a href="#">VAR</a> statement.
<b>CHOICES</b>	Modifies a choice list variable with the <a href="#">RANGESET</a> statement.
<b>CLASS</b>	The CLASS statement defines the Java class which does the real work. Customer supplied device drivers do not require this statement.
<b>COMMENT</b>	The <a href="#">COMMENT</a> statement defines an identification for the driver which is reported to the user interface.
<b>COPY</b>	The <a href="#">COPY</a> statement copies a single parameter or a complete device setup from one device to another.
<b>CUT</b>	Cuts a number of characters in the <a href="#">INPUT</a> statement.
<b>CYCLE</b>	Defines the refresh cycle for a variable in the <a href="#">VAR</a> statement.
<b>DELAY</b>	The <a href="#">DELAY</a> statement pauses the driver execution for a certain time.
<b>DISABLED</b>	Marks a variable to be initially disabled in a <a href="#">VAR</a> statement or disables a variable with the <a href="#">RANGESET</a> statement.
<b>DRATE</b>	The <a href="#">DRATE</a> statement computes an interface data rate from a symbol rate and modulation/encoding parameters.
<b>ENABLED</b>	Enables a variable with the <a href="#">RANGESET</a> statement.
<b>FLOAT</b>	Defines a variable to be a floating point number in a <a href="#">VAR</a> statement.
<b>FLOAT16</b>	Writes or reads a 16 bit floating point number with a <a href="#">WRITE</a> or <a href="#">READ</a> statement.
<b>FLOAT32</b>	Writes or reads a 32 bit floating point number with a <a href="#">WRITE</a> or <a href="#">READ</a> statement.
<b>FLOAT64</b>	Writes or reads a 64 bit floating point number with a <a href="#">WRITE</a> or <a href="#">READ</a> statement.
<b>FMT</b>	Formats a number to a string in the <a href="#">PRINT</a> statement.
<b>GET</b>	Marks a procedure to be a GET-type one in the <a href="#">PROC</a> statement.
<b>GOTO</b>	The <a href="#">GOTO</a> statement jumps to a <a href="#">label</a> in a procedure.

<b>HEX</b>	Interprets a string as a hexadecimal number in an <a href="#">INPUT</a> statement or defines a variable to be a hexadecimal number in a <a href="#">VAR</a> statement.
<b>IF</b>	The <a href="#">IF</a> statement conditionally executes the following statement.
<b>INCLUDE</b>	The <a href="#">INCLUDE</a> statement reads another source file.
<b>INIT</b>	Tells the driver to initialize a variable at power up with a certain value in the <a href="#">VAR</a> statement.
<b>INPUT</b>	The <a href="#">INPUT</a> statement reads a message from the device and parses the reply as text.
<b>INT16</b>	Writes or reads a 16 bit integer number with a <a href="#">WRITE</a> or <a href="#">READ</a> statement.
<b>INT32</b>	Writes or reads a 32 bit integer number with a <a href="#">WRITE</a> or <a href="#">READ</a> statement.
<b>INT64</b>	Writes or reads a 64 bit integer number with a <a href="#">WRITE</a> or <a href="#">READ</a> statement.
<b>INT8</b>	Writes or reads an 8 bit integer number with a <a href="#">WRITE</a> or <a href="#">READ</a> statement.
<b>INTEGER</b>	Defines a variable to be an integer number in a <a href="#">VAR</a> statement.
<b>INVALIDATE</b>	The <a href="#">INVALIDATE</a> statement marks a variable to be re-read in this or the next driver cycle.
<b>LOG</b>	The <a href="#">LOG</a> statement writes a message into the event log.
<b>MAX</b>	Modifies the upper range limit of a numeric variable with the <a href="#">RANGESET</a> statement.
<b>MIN</b>	Modifies the lower range limit of a numeric variable with the <a href="#">RANGESET</a> statement.
<b>OBJECT</b>	Defines a variable to be a 'object' in a <a href="#">VAR</a> statement.
<b>OFFSET</b>	Adds an offset to as value in a <a href="#">PRINT</a> or an <a href="#">INPUT</a> statement.
<b>PRINT</b>	The <a href="#">PRINT</a> statement composes a text message/command and sends this to the device.
<b>PROC</b>	The <a href="#">PROC</a> keyword starts a procedure definition.
<b>PROTOCOL</b>	The <a href="#">PROTOCOL</a> statement defines the (preferred) communication protocol a driver is designed to use.

<b>PROTOCOLPARAMETER</b>	The <a href="#">PROTOCOLPARAMETER</a> statement defines parameters to be passed to the communication protocol.
<b>PUT</b>	Marks a procedure to be a PUT-type one in the <a href="#">PROC</a> statement.
<b>PUTPRIORITY</b>	Defines that the driver shall use the put priority execution scheme for PUT/GET procedures. See Topic <a href="#">Tweaking the processing sequence</a> for details.
<b>RAISECOMMFAULT</b>	The <a href="#">RAISECOMMFAULT</a> statement makes the driver to raise a communication fault.
<b>RANGESET</b>	The <a href="#">RANGESET</a> statement modifies range properties of a variable during runtime.
<b>READ</b>	The <a href="#">READ</a> statement gets a message from the device and parses the data as a binary data structure.
<b>READHEX</b>	The <a href="#">READHEX</a> reads a message from the device and formats the data as a hex dump string.
<b>READONLY</b>	Marks a variable to be read only in a <a href="#">VAR</a> statement or makes the variable read only with the <a href="#">RANGESET</a> statement.
<b>READWRITE</b>	Makes a variable writable with the <a href="#">RANGESET</a> statement.
<b>SAVE</b>	Tells the driver do save a variable's value on disk in the <a href="#">VAR</a> statement.
<b>SCALE</b>	Multiplies a value with a scale factor in a <a href="#">PRINT</a> or an <a href="#">INPUT</a> statement.
<b>SEND</b>	The <a href="#">SEND</a> statement sends a parameter message to another device.
<b>SET</b>	The <a href="#">SET</a> statement assigns a value to a variable.
<b>SETUP</b>	Makes a variable appear in the setup menu for the device <a href="#">VAR</a> statement.
<b>SKIP</b>	Skips whitespace characters in the <a href="#">INPUT</a> statement.
<b>SRATE</b>	The <a href="#">SRATE</a> statement computes a symbol rate from a data rate and modulation/encoding parameters.
<b>SUBROUTINE</b>	Marks a procedure to be a subroutine when following the <a href="#">PROC</a> keyword.
<b>SUBSCRIBE</b>	Marks a procedure to be called if the value of a variable changes when following the <a href="#">PROC</a> keyword.

<b>SYNC</b>	Syncs a variable, updates the value to be commanded with the value recently read back from the device.
<b>TABLE</b>	The <a href="#">TABLE</a> statement defines a translation table.
<b>TEXT</b>	Defines a variable to be plain text in a <a href="#">VAR</a> statement.
<b>TRM</b>	Cuts a string up to a given termination character in the <a href="#">INPUT</a> statement.
<b>VAR</b>	The <a href="#">VAR</a> statement defines a driver variable.
<b>WATCH</b>	Binds a procedure to one or more variables with the <a href="#">PROC</a> statement.
<b>WRITE</b>	The <a href="#">WRITE</a> statement composes a binary message/command and sends this to the device.
<b>WRITEHEX</b>	The <a href="#">WRITEHEX</a> statement converts a hex dump string to a binary message, sends this to the device.
<b>XLT</b>	Translates a character string through a table in a <a href="#">PRINT</a> or an <a href="#">INPUT</a> statement.

### 1.10.2 Protocol definition keyword reference

The following table lists the keywords recognized in protocol definition files in alphabetic order.

<a href="#">ADDRESS</a>	Outputs the device address.
<a href="#">ADDRESS</a>	Reads the device address.
<a href="#">CHAR</a>	Outputs a single character.
<a href="#">CHAR</a>	Reads a single character.
<a href="#">CHECKSUM</a>	Outputs a message checksum.
<a href="#">CHECKSUM</a>	Reads a message checksum.
<a href="#">CLASS</a>	Specifies the underlying protocol class.
<a href="#">COMMENT</a>	Defines the protocol identification string.
<a href="#">DATALENGTH</a>	Outputs a data length field (binary).
<a href="#">DATALENGTH</a>	Reads a data length field (binary).
<a href="#">HEXLENGTH</a>	Outputs a data length field (hex).
<a href="#">HEXLENGTH</a>	Reads a data length field (hex).

<a href="#">PROTOCOLPARAMETER</a>	Defines parameter to be passed to the communication protocol
<a href="#">RECEIVE</a>	Starts the receive protocol step specification.
<a href="#">SEQUENCE</a>	Outputs a binary message sequence number.
<a href="#">START</a>	Reads a message start character.
<a href="#">STRING</a>	Reads a terminated string as the user data
<a href="#">TRANSMIT</a>	Starts the transmit protocol step specification.
<a href="#">USERDATA</a>	Outputs the user data.
<a href="#">USERDATA</a>	Reads the user data.

### 1.10.3 Help file keyword reference

**Attention!** Please remember that this is deprecated and the new file format ist Markdown.

#### deprecated:

The following table lists the 'dot commands' recognized by the help compiler in alphabetic order. They all are explained in the chapter ' [Help file format](#) '.

<b>.ul</b>	Starts a bullet list.
<b>.ol</b>	Starts an ordered (numbered) list
<b>.li</b>	Starts a list item.
<b>.br</b>	Inserts a line break.
<b>.eul</b>	Closes a bullet list.
<b>.eol</b>	Closes an ordered (numbered) list
<b>.topic</b>	Defines a topic header.
<b>.p</b>	Inserts a paragraph break.
<b>.br</b>	Inserts a line break.
<b>.dot</b>	Inserts a '.' (dot) character.
<b>.tl</b>	Starts a table definition
<b>.ts</b>	Starts the first cell of a table row
<b>.tc</b>	Separates a table cell from the next one in the same row

<b>.te</b>	Marks the end of the last table cell in a row.
<b>.etl</b>	Closes the table definition.
<b>.i</b>	Includes an image.
<b>.pre</b>	Starts pre-formatted text
<b>.epre</b>	Ends pre-formatted text

### 1.10.4 Debugging with Terminal Session

The **sat-nms** client provides the *terminal session window* which opens a TCP connection to Port 2000 of the MNC server. This debugging port provides several command to intercept devices and device communication and to manipulate parameters. It is also possible to access this port with any terminal program (e.g. putty with connection type `raw`).

#### 1.10.4.1 Terminal Session Commands

Enter **?** on the command prompt to get the following short help overview:

```
---> ?
general information:
  server / faults / msg-count
messages:
  get <msg-id> / set <msg-id> <v> / loop <msg-id> <v>
message distributors:
  cdis (debug|nodebug)
  ddis (debug|nodebug)
device threads:
  threads / thread <n>
client peers:
  peers / peer <n> (view|verbose|silent|kill)
system:
  restart / shutdown / q
debug log:
  limit / nolimit
```

#### server

Shows information of **sat-nms** MNC server process including:

- state: current server state. R = normal operation
- clients: number of connected clients
- d-threads: number of devices threads (see **threads** )
- msg-rate: The **server** command outputs the current rate of updated driver variables. The rate is determined over an interval of 5 seconds. Variable update calls are counted and divided by the time that has passed since the rate was last determined. The value is

a bit too high in terms of communication to real equipment, because also the (internal) messages between logical devices are counted as well

Example:

```
---> server
state:    R
clients:  3
d-threads: 54
msg-rate: 505.69 msgs/sec
```

## faults

Shows a list of all devices (one per line) and the current summary faults or warning state. Devices which are ok, just show no message text.

Example: where `HPA-2` is in WARNING state and `HPA-3` in FAULT state

```
...
HPA-1:
HPA-2: WARNING
HPA-3: Summary FAULT
S1:
S2
```

## msg-count

Outputs the absolute counter readings of the messages distributed by the MNC server since startup. Starts with a high number, because all the range messages are counted at the start.

Example:

```
distributed messages
from devices 2167806180
to devices 2261258
```

## cds debug / cds nodebug

Enables or disabled debug output on stdout (usually found in `/home/satnms/.panic.log` ) for the client distributor. This show messages going from server to the clients (including other devices).

**Attention : Enabling this output creates huge amount of data which on the disk. Do not forget to disable after testing.**

Example: A beacon level (parameterId `BCRX.power` ) is send to client and to a File-Recorder logical device.

```
BCRX.power distributed to satnms3.vlc.ClientPeer (176)
BCRX.power distributed to satnms3.device.FileRecorder@57576068
```

## ddis debug / ddis nodebug

Enables or disabled debug output on stdout (usually found in `/home/satnms/.panic.log` ) for the device distributor. This show messages going to devices (including logical devices).

**Attention : Enabling this output creates huge amount of data which on the disk. Do not forget to disable after testing.**

Example: A BUC is commanded to transmit (parameterId `BUC-1.tx.on` ). This messages usually have their origin from user input.

Example:

```
BUC-1.tx.on distributed to tx.on ON
```

## threads

Shows all list of all threads (which corresponds to the interface definition) and the current thread state. A threads / interface can have 1 or more attached devices.

Example:

```
---> threads
0 | DT-null    (2 devices)
1 | R DT-null  (1 devices)
2 | I DT-null   (6 devices)
3 | I DT-10.10.1.13:4001(1 devices)
4 | I DT-ttyS11 (1 devices)
5 | I DT-ttyS0  (1 devices)
```

## thread `n`

Show details for thread `n` . This includes (for each device attached to this thread/interface)

- device name
- device class: a Java class name if the device is hardcoded or contains hardcoded function, or `satnms3.device.UniversalDriver` if the driver is written completely in the **sat-nms** Universal Device Driver Language
- driver name
- measured cycle time: minimal/average/maximal measured cycle time in msec. This show how long the device needs to go through all GET (and PUT) procedures. It includes the idle time as configured for this interace (usually 1000 msec)
- protocol name
- protocol class: a Java class name if the protocol is hardcoded or `satnms3.proto.TTYProtocol` if the protocol is written in the **sat-nms** Universal Device



#### Driver Language

- interface
- response time: minimal/average/maximal measured response time for messages send to a (real) device in msec.
- thread state

Example: **sat-nms** LBRX Beacon Receiver with HTTP protocol

```
---> thread 19
thread 19 DT-null (I)
device 1
  device name: BCRX-1M8
  device class: satnms3.device.UniversalDriver
  driver name: SatService-Beacon-Receiver 3.03 200430
  measured cycle time: 1124/1254/1378 msec
  protocol name: HTTP 1.01 090728
  protocol class: satnms3.proto.HTTPProtocol
  interface: null
  response times: 120/157/215 msec
```

#### peers

Show list of all connected clients and their current state

Example: 3 connected clients from different hosts

```
---> peers
0  RI 10.10.1.143
1  RI 192.168.2.22
2  RI 192.168.2.34
```

#### peer **n** (view|verbose|silent|kill)

peer function	description
verbose	print all message between client and server
silent	disable verbose output
kill	close connection, drop client
view	show peer details

Example: shows details for client 1

```

---> peer 0 view
ClientPeer 0:
id:          50
destination: 10.10.1.143
input state:  R
output state: I
queue fill:   0
uptime:       16d21h
msgs received: 297387
msgs sent:    2637717

```

Client peer input and output states:

state	description
I	Initializing
R	Reading / waiting for Data
P	Processing Data
D	Dead (Streams closed)
O	Outputting (sending Data)
S	Stopping

Example: enabled client/server verbose output where a new beacon level is send to the client and a keep alive message is exchanged between client and server (ping)

```

sent BCRX.power -71.56
sent @ping. Fri Jul 30 15:09:47 UTC 2021
got  @cpng.

```

## restart

restart MNC server, similar to `satnms-stop; satnms-start` on command line or the restart button the client UI.

## shutdown

shutdown MNC server. similar to `satnms-stop` on command line

## q

close terminal session

## limit / nolimit

disabled log rotation of `/home/satnms/debug.log` .

**Attention : Enabling this output creates huge amount of data which on the disk which will cleaned up automatically. Do not forget to disable after testing.**

#### 1.10.4.2 Device Message Commands

Within the terminal session it is possible to read and write every parameter of all devices. A typical use case is to mock parameters of equipment which is currently not connected.

command	description
get msg-id	show the current value of a parameter specified by msg-id
set msg-id v	set the value v of parameter specified by msg-id . This injects a message to the event distributor in the same way as when a user command a value from a client
loop msg-id v	set the value v of parameter specified by msg-id . This injects a message to the event distributor that looks like it was received from the equipment (response on polling)

#### get msg-id

The get command returns all parameters starting with the string in msg-id . If you specify a name which does not exist it returns an empty line. If msg-id matches more than one parameter multiple lines will be return. This allows for example to get all parameter from one device with get BUC-1 or a subset of parameters like get BUC-1.faults .

Normally 2 lines return per parameters:

- the parameter itself with the current value
- the range setting of this parameter (marked with [msg-id].R ) which describes the type range and range.
- Disabled parameters (marked with DIS ) or parameters which where never initialized will only result in one line with the range message.

Example: Parameter BUC-1.tx.on which is a CHOICE parameter with possible values ON and OFF, currently set to ON

```
---> get BUC-1.tx.on
BUC-1.tx.on.R EnumRange (ON|OFF)
BUC-1.tx.on OFF
```

The get command recognizes a parameter -r which causes the command to suppress all range messages which otherwise would be included in the list.

```
---> get -r BUC-1.tx.on
BUC-1.tx.on OFF
```

The range message belonging to this parameter is not shown.

**set** `msg-id` `value`

The 'set' command does not produce any output (except logParameterChanges is enabled for the device). It performs also no range checking and setting values of an unknown messageld will be silently ignored.

In case of a communication fault, the driver will stop after the first INPUT statement which does not get answered by the equipment and starts from the beginning. Therefore you will not see any outgoing (PUT) commands even if you set a parameter with `set` until you stop the driver cycle aka polling (see below).

Example: switch BUC transmit to ON

```
---> set BUC-1.tx.on OFF
000000 2021-07-30 16:07:50 2021-07-30 16:07:50 I BUC-1 tx.on changed to 'OFF' by debug
```

**loop** `msg-id` `value`

A loop commands simulates that the device receives a message from equipment. So it looks like, that the equipment response to polling. With this command you can mock e.g. read-only values to display something useful even if you have no equipment connected.

But remember, values set with the `loop` command will be overwritten by next successful polling result (if there is communication to the equipment)

Example: set the output power of an BUC to 162 (Watt). On first `get` BUC-1.meas.fwdPwr is not initialized, but after `loop` ing 162 to the device it show this new value on the next `get`

```
---> get BUC-1.meas.fwdPwr
BUC-1.meas.fwdPwr.R IntegerRange R/O (0 .. 0)

---> loop BUC-1.meas.fwdPwr 162

---> get BUC-1.meas.fwdPwr
BUC-1.meas.fwdPwr 162
BUC-1.meas.fwdPwr.R IntegerRange R/O (0 .. 0)
```

## verbose mode

Enable *verbose mode* for a device to follow all communication between MNC server and equipment *Verbose mode* could be enabled in the device setup page of each device or in the terminal session with `set [DEVICENAME].verbose true` .

Example: Beacon Receiver with HTTP protocol. Parameter BCRX.threshold is read from the device with the command `/rmt?thr=?`

```
---> set BCRX.verbose true
tx(BCRX): /rmt?thrh=?
rx(BCRX): thrh=-90.00
BCRX.threshold=-90.00
```

In addition you can enable *dump mode* for a device to print full communication between MNC's server and equipment even if it does not match with the required protocol structure with `set [DEVICENAME].dump true` .

**Attention : Enabling this output creates huge amount of data which on the disk so do not forget to disable after testing.**

### stop polling cycle

You can stop the driver cycle with the following command in the terminal session:

`set [DEVICENAME].cycle true` . Re-enable polling with `set [DEVICENAME].cycle false` or a restart of the MNC server.

Example: disable polling and set a value on a not connected device. After disabling polling the device tries one more time to communicate ( `tx(BUC-1): GET ...` ) but does not expect a response anymore. Device state switches to Communication OK.

```
---> set BUC-1.cycle false
tx(BUC-1): GET 1.3.6.1.4.1.2566.127.1.5.10.1.1.14.1.2.0 (FAILED)
000000 2021-07-30 15:52:20 2021-07-30 15:52:20 | BUC-1 past 0m17s Communication OK.
000000 2021-07-30 15:52:20 2021-07-30 15:52:20 | BUC-1 OK.
```

Now you can issue a `set` command (or send commands via Client UI) and the corresponding PUT procedures will be executed. Of course you will still not get any response from the equipment, so the Communication Fault will be issued.

```
---> set BUC-1.tx.on OFF
000000 2021-07-30 15:57:46 2021-07-30 15:57:46 | BUC-1 tx.on changed to 'OFF' by debug
tx(BUC-1): SET 1.3.6.1.4.1.2566.127.1.5.10.1.2.1.3.1.0 | 1 (FAILED)
000000 2021-07-30 15:57:55 2021-07-30 15:57:55 | BUC-1 past 2m57s Communication OK.
000000 2021-07-30 15:57:55 2021-07-30 15:57:55 | BUC-1 OK.
```

### 1.10.5 Global faults

The M&C software knows two global faults flags indicating fault conditions which are not linked to devices or subsystems:

#### SYSTEM.fault

indicates some sort of general fault, mostly faults occurring during the initialization phase of the M&X. SYSTEM.fault may be one of "OK." and "Summary FAULT". Which condition caused

the fault is listed in the event log. A "RESET" to the SYSTEM device clears this fault.

### **SUMMARY.fault**

summarizes the faults of all devices in the M&C. SUMMARY.fault knows two states, 'FAULT' and 'OK.'. It turns to 'FAULT' if at least one device in the M&C shows a fault.