OpenTPL
Open Transfer Protocol Language

A protocol for client-server based exchange of data and commands over a TCP/IP network connection

Protocol specification and implementation guide line

Version 2.1

Document ID:

`opentpl:spec-en`

Revision 32 as of December 19, 2014 16:07:08

Revision history:

Date	Version	Changes
02.12.2002		Initial Version (dp)
05.11.2003		Some additions (mr)
13.02.2004		Revision of `DATA` syntax, added TLS (mr,dp)
10.09.2004	2.0	Finalized revision (dp)
16.08.2005		Revision for assimilation of Java/C++ implementations (mr)
07.08.2012		Revision, some unused protocol details have been removed, updated layout (mr,ms)
18.11.2013	2.1	Add server based authentication management, connection informations (ms, mr)
23.06.2014		Finalization of authentication management (mr)

OpenTPL — Open Transfer Protocol Language (A protocol for client-server based exchange of data and commands over a TCP/IP network connection)

Version 2.1

Document ID: `opentpl:spec-en`

Revision 32 as of December 19, 2014 16:07:08 by Michael Ruder

`opentpl_spec-en.lyx 32 2014-12-19 16:07:08Z ruder`

Contents

1. Preface

The main motivation to define a new command and data exchange protocol was the need for a simple, yet quite powerful, human readable ASCII text protocol that can be used to control various kinds of hardware such as telescopes and their related components. Currently, this is the only application of OpenTPL but as it is designed to be as flexible as possible, a lot of other applications are imaginable.

The OpenTPL protocol can be used for free by anybody for any purpose and the authors encourage anybody interested in implementing an OpenTPL server or client for their own purposes to do so.

This document is aimed to address two issues: First, it shall give a detailed protocol definition and second, it shall give some guide lines with respect to a possible implementation.

2. Basics

2.1. Overview

- All communication (i.e. data, command and status/error exchange) is done over a TCP/IP socket connection. The used port can be defined by the user.
- For debugging reasons, OpenTPL is a human readable ASCII clear text protocol.
- All OpenTPL commands, module names and variables are case insensitive and all server replies can be expected to be in uppercase. However, string variables store the data case sensitive.
- All lines sent by the server are terminated by LF, some implementations may send a CR before the LF. The server accepts both types for incoming lines.
- OpenTPL supports a simple access control scheme with privilege levels based on username/password authentication.
- Most implementations will include the TLS protocol to encrypt the entire communication (making the username/password login safe) and to do a certificate based authentication of the communicating hosts. See section 3.1.1.
- OpenTPL is designed as client-server protocol. This means that the protocol is asymmetric: the client sends commands and data and the server replies back with status, data or events.
- Both the server and the client can send at any time. The client must be aware that status/event messages can come in at any time. Also answers from concurrent running commands can come in in any order.
- All functionality offered by the server is mapped to variables. When the client reads from or writes to these variables, the corresponding functions are executed by the server.
- Variables can be grouped in modules. This allows functional groups to enhance hierarchical structures. There is a special, standardized module `SERVER` which contains information about the server and the host it is running on.

2.2. Conventions in this document

This manual uses the `typewriter font` whenever OpenTPL protocol text is shown. Text that is enclosed in <...> stands as a placeholder. So `AUTH PLAIN` <*username*> <*password*> means that the actual username and password and not the string <username> is expected by the server. Sometimes parameters are optional. This is shown by putting the text in [...]. If these brackets are used in the OpenTPL protocol itself, they will be typeset as `<...>` and `[...]`. Special design suggestions will be marked with **Implementation Note** at the beginning of a paragraph.

3. Connection setup

After opening a TCP socket connection to the server, the server will send a greeting message of the format

`TPL2` <*protocol version*> `CONN` <*number*> `AUTH` [<*method*>`,`<*method*>`, ...`]

`ENC` [<*algorithm*>`,`<*algorithm*>`,...`] [`MESSAGE` <*message*>]

The message consists of several keywords and values to those keywords. Keywords and their values are separated by spaces. No spaces may occur in the values, beside the value to the `MESSAGE` keyword. The order of the keywords needs to be exactly as shown in the above example message.

The following keywords may appear in the greeting message:

`TPL2` The is the first keyword of the greeting message. Its value is the highest supported protocol version in the form major.minor, e.g. "`2.0`".

`CONN` The value to this keyword is an unsigned 32 bit connection number (between 0 and 4294967295) that uniquely identifies the current connection.

`AUTH` This value lists the supported authentication algorithms (see section 3.2) supported by the server in a comma separated list that does not contain any spaces. The list can be empty if the server is configured for implicit authentication (see 3.2).

`ENC` This value lists the supported encryption algorithms (see section3.1) supported by the server. The list can be empty if the server does not support encryption.

`MESSAGE` The value of this keyword can be any string that will continue to the end of the line and may contain spaces. It can be omitted completely.

3.1. Encryption

The encryption has to be started directly after connection setup, even before authentication takes place.

To switch into encrypted mode, the special command

`ENC` <*method*> [<*parameter*> <*parameter*> `...`]

is used. Note, that there is no command id. Not all encryption methods need <*parameter*>s and their meaning varies.

If the method is recognized and supported, the server will then answer with

`ENC OK`

and switch into encrypted mode. So this is the last unencrypted line the server will send. An encrypted connection stays encrypted until it is closed. There is no command to turn off the encryption again.

In case of any syntax error the message

`ENC ERROR`

is sent. Is the specified method unsupported, the reply will be

`ENC UNSUPPORTED`

and in both cases the connection remains unencrypted.

3.1.1. `TLS` encryption

This encryption method uses the Transport Layer Security protocol (the successor of SSLv3) as described in RFC 3546. It is invoked with

`ENC TLS`

and does not need any special parameters as these are negotiated by the TLS protocol.

3.2. Authentication

If the server lists authentication algorithms in the greeting line, the client needs to authenticate itself using the

`AUTH` <*method*> [<*parameter*> <*parameter*> ...][<*read level*> <*write level*>]

command.

Without completed authentication, only encryption (`ENC`), authentication (`AUTH`) and disconnection (`DISCONNECT`) commands are accepted by the server. Any other command is rejected with the error `UNAUTHENTICATED`. Also, no `EVENT` messages are forwarded to the connection before authentication.

If no `AUTH` methods are offered in the greeting message (see section 3), the server will immediately send the `AUTH OK` message (see below) and the client must not send any authentication commands.

Depending on the mechanism used, the meaning of the optional <*parameter*>s differ and is explained in the following sections. It is also possible that some mechanisms require multiple steps.

As the last two parameters the user may add its desired read and write level (see below). If it is higher than the lowest level granted to the user by the server configuration, this level comes into effect, otherwise the level from the configuration is used (and no error is generated). This allows the user to login with reduced privileges.

If the authentication was successful, the server will reply with

`AUTH OK` <*read level*> <*write level*>

These are the effective read and write levels. With these levels the client can check if it is allowed to read from or write to a variable before actually accessing it (see sections 4.2, 4.3 and 8.4).

If the submitted credentials were not valid, the message

`AUTH FAILED`

is sent. For security reasons this message can be delayed by a few seconds. It is also possible that the server will close the connection after one or more unsuccessful authentication attempts. In case of any syntax error the message

```
AUTH ERROR
```

is sent and if the requested mechanism is not supported, the message

```
AUTH UNSUPPORTED
```

is sent.

It is possible to disable insecure mechanisms in case of unencrypted connections. In this case, the server will send

```
AUTH DISABLED
```

3.2.1. `PLAIN` authentication

In this case the client sends

`AUTH PLAIN` <*username*> <*password*>[<*read level*> <*write level*>]

to the server. Both <*username*> and <*password*> are sent in clear text and need to be enclosed in double quotes.

> **Implementation Note** All server implementations should implement this authentication form as it is easy to use during testing and debugging. It should however be possible to disable this potentially insecure authentication method either on all or only on unencrypted connections. If it is disabled only on unencrypted mechanisms, the server should still announce it in the greeting message but generate a DISABLED (see section 3.2) error when it is used on unencrypted connections.

3.2.2. `CERT` authentication

For this method, the clients sends

`AUTH CERT`[<*read level*> <*write level*>]

to the server. An encrypted connection is mandatory for this authentication method and not all encryption methods may work together with this authentication.

If the client certificate that was presented during encryption setup is considered valid by the server, the authentication will be successful. No further parameters are needed.

3.3. Terminating a connection

To cleanly terminate the current connection, the client can send

```
DISCONNECT
```

at any time (i.e. also before authentication or encryption was performed). The server will answer with a

```
DISCONNECT OK
```

immediately before closing the connection.

Depending on the connection setting `ABORT_ON_DISCONNECT` (see section 9.1) all running commands of that connection might be aborted afterwards.

It is strongly recommended, that clients cleanly disconnect using the `DISCONNECT` command to avoid unnecessary resource allocation by stalled connections.

4. Commands

For a maximum of flexibility, all functionality of an OpenTPL server is realized with a hierarchical structure of objects (see section 4.1). There are modules and variables. Both have properties which allow the client to collect information about the server. Inside the server these variables can have a callback function which is executed by reading from or writing to these variables. After a value is written to a variable, the callback function is called and can carry out any task e.g. send this value to some hardware driver and move a motor. Accordingly, by reading from this variable, the callback function is called first and can retrieve some position or status data from the hardware which is then sent to the client. Due to this fact, the execution of commands may take considerable amounts of time and the client may not misinterpret this as timeout condition.

> **Implementation Note** All server implementation should be able to apply a configurable limit to the number of the concurrently running callbacks and the queued commands. In the latter case, an error is produced when a new command is issued (see section 4.2, 4.3). Additionally, each callback can either be reentrant, e.g. allow multiple instances running at the same time or not. This behavior can be queried with the properties of the variable that offers the callback.

To read from variables, the server offers the `GET` command (see section 4.2). Accordingly, for writing to variables the `SET` command (see section 4.3) is offered.

All commands are carrying a unique unsigned 32 bit command id between 1 and 4294967295. The server will prepend this id to any lines it will send in response to the command. Issued commands are executed by the server in parallel, or, if too many commands arrive, are queued until execution slots are available. The response lines of different commands may therefore be intermixed. However, for each command the order of its response lines will always be as described in the following sections. The command id may not be reused until the command is complete, e.g. the final response line was sent by the server.

The client can send new commands at any time. If too many commands are queued at the server, the server may however reject additional commands, until the queue gets shorter again.

In addition to the response lines that belong to an issued command, the server may send other lines: General server error messages, broadcasted `EVENT`s that do not refer to any running command or messages that refer to a command that contained an invalid or omitted command id will be answered with the special command id `0`. `EVENT`s that refer to a running command will carry its number. If the command was issued on a different connection they will carry a 64 bit command id, where the upper 32 bits carry the number of the connection (see section 3).

> **Implementation Note** Internally any server or client implementation should keep track of its commands by using the above described extended 64 bit command id.

To terminate the execution of a running `GET` or `SET` command, the `ABORT` command is offered. It will try to cleanly terminate the running command, either by asking its callback function to end

or by removing the command from the queue if it's execution did not already start, which may or may not succeed.

Any command is first acknowledged by the server by a

<*cmdID*> `COMMAND` <*state*>[[<*message*>]]

line, where the following states may occur for any command:

State	Description
`OK`	The command is valid and will now be queued for execution.
`ERROR UNAUTHENTICATED`	The user needs to authenticate first (see section 3.2).
`ERROR IDBUSY` <*err cmdID*>	The <*err cmdID*> is in use by a currently executing command. In this case, <*cmdID*> is set to 0.
`ERROR IDRANGE` <*err cmdID*>	The <*err cmdID*> exceeds the allowed range. In this case, <*cmdID*> is set to 0.
`ERROR SYNTAX`	Command syntax error. In this case, <*cmdID*> may be set to 0.
`ERROR TOOMANY`	Too many commands are already in the queue.
`ERROR UNKNOWN`	Unrecognized command. In this case, <*cmdID*> may be set to 0.

The optional <*message*> may explain the error further and is for logging purposes only. If <*state*> was `OK`, the server will queue the command for execution (or immediately execute it, if the server load is low enough).

After the command specific output, the server sends a final line, indicating the end of the command:

<*cmdID*> `COMMAND` <*state*>[[<*message*>]]

Here, the following states may occur for all commands:

State	Description
`COMPLETE`	The command was executed (even so there might have been some errors reported in the command specific output).
`ABORTEDBY` <*abort cmdID*>	The command has been aborted using the `ABORT` command and aborted cleanly. <*abort cmdID*> is the id of the `ABORT` command and can be an extended id if it was issued from another connection or a normal id if it was issued from the client's connection.
`FAILED`	The command could not be executed at all (in case of a `COMMAND ERROR ...` in the acknowledge message).

After this message was sent, the <*cmdID*> is no longer in use and can be used again. Consequently the server will not generate any more messages with this number until another command uses this id again.

4.1. Object specification

Both the `GET` and the `SET` commands work on objects and therefore need one or more object specifications as parameters.

Objects in OpenTPL consist of a hierarchical path of the form <*object*>[.<*subobject*>[...]], where the <*(sub)object*> can be a module or, if it is specified last, a variable. In the following sections, this path is abbreviated with <*object*>.

4.1.1. Arrays

Some objects can be arrays. In this case, the element index is appended in the form <*subobject*>`[`<*index*>`]`. It is also possible to specify more than one index, some examples are shown in the following table:

Index specification	Description
`[`<*index*>`]`	Single element
`[`<*index*>`-`<*index*>`]`	Range of elements (including both limits)
`[`<*index*>`[,`<*index*>`[,...]]]`	Several elements
`[`<*index*>`,`<*index*>`-`<*index*>`,`<*index*>`]`	Both types mixed (any mixture is possible)

If several arrays are part of the hierarchical structure, only one of the indexes can specify multiple elements, otherwise the server will return an `INVALID` error.

4.1.2. Slices

If the object is a string variable, it is possible to access only part of it (a so called slice) by appending a byte range: <*object*>`{[`<*begin*>`]:[`<*end*>`]}`. This range includes both limits and if <*begin*> or <*end*> is omitted the beginning or end of the variable is used. Only one range may be specified. The first character has the number 0. If multiple array elements were specified (see section 4.1.1) the slice specification is applied to all these elements.

4.1.3. Properties

For accessing properties of the object rather than the object itself, the name of the property is appended with an exclamation mark: <*object*>`!`<*property*> (see section 8 for a list of supported properties). Only if the object is a variable, this property specification can be omitted to access the value of the variable. For all other object types, accessing only the object without a property specification will result in an `INVALID` error.

4.1.4. Examples

The following table gives an overview of different types of object specifications:

Object specification	Description
<*module*>`!`<*property*>	Module or module array properties
<*module*>`[`<*index*>`-`<*index*>`]!`<*property*>	Properties of several elements in an array

Object specification	Description
<*module*>.<*variable*>	Value of a variable
<*module*>.<*variable*>!<*property*>	Variable or variable array properties
<*module*>.<*variable*>[<*index*>]!<*property*>	Properties of a variable in an array
<*module*>.<*variable*>[<*index*>]	Value of a variable in an array
<*module*>.<*variable*>[<*index*>]{<*begin*>:<*end*>}	Slice of a string variable

4.1.5. Numerical access

Beside using symbolic names for object specification, each object can be accessed by its internal reference numbers in angle brackets: `<<`*number*`>>`.

> **Implementation Note** All server implementations need to assign these numbers to all objects and subobjects during startup, starting from zero up to the count of subobjects minus one. These numbers must not change during server run time but they may change if the server is restarted, especially if the configuration files were modified.

This access mode allows the client to search for all objects of the server using properties like `!MEMBERS` and `!NAME` (see section 8):

<*cmdID*> `GET <5>.<3>!NAME`

4.2. `GET` — Retrieving data

Reading from variables and properties is done with the `GET` command. It has the following syntax

<*cmdID*> `GET` <*object*>[`!`<*property*>][`;`<*object*>[`!`<*property*>][`;...`]]

For possible object specification please refer to section 4.1. It is possible to specify more than one object and each of these object specifications can itself refer to multiple objects. The specified objects will be retrieved sequentially, allowing control over the execution order. (If parallel execution is desired, multiple `GET` commands can be sent instead).

The server will send an immediate acknowledge (or error) message after parsing the command as described in section 4.

For objects of type variable with no property specified, the read level of the variable is then compared with the read level of the authenticated user. If access is granted, the callback function of the variable is executed (if it has one) and then the variable contents is sent to the client. If multiple objects are requested, the callback functions of all requested objects are called sequentially (e.g. for each accessed array element).

For all other object types, only properties can be requested. Access to properties will never trigger callback functions and is possible regardless of the user's read level.

All data is returned in the form

<*cmdID*> `DATA INLINE` <*object*>`=`<*value*>[`,`<*value*>[`,...`]]

<*value*> is always returned as text, strings are enclosed in double quotes, special characters are escaped (see section 7.1). The object specification <*object*> is returned exactly as it was sent in the `GET` command, i.e. if access was performed by the "`<...>`" syntax or multiple array indexes were specified, the reply will also use this syntax. This allows easier client design, since the client does not need to fully parse the server answer. If multiple array indexes were accessed, these are returned comma-separated.

If a slice was specified for a string variable and the end lies outside the data of the variable, the returned amount of data will be shorter than the specified slice length. If even the slice start lies outside the data of the variable, an empty string will be returned (Exception: if the string variable contains a NULL value, any slice will also return NULL).

To indicate the return of a non-initialized variable that has no valid value, the special keyword NULL will be returned. To avoid confusion with the string "NULL" this keyword will not be enclosed in double quotes. If errors occurred during retrieval or during the execution of the callback functions, the error keywords (see below) are returned instead of a value and will also not be enclosed in double quotes to avoid confusion with strings.

The following tables show the errors, that can occur with DATA. If the error occurs before the callback would have been called, this is indicated by "no callback".

Error	Description
BUSY	The callback is non-reentrant and already running (no callback).
DENIED	Read access is not granted (no callback).
DIMENSION	The index is out of bounds (no callback).
FAILED <*code*>	The callback function returned a fatal error (for additional information a <*code*> can be returned).
INVALID	A module was specified (without a property) (no callback).
LOCKEDBY <*lock cmdID*>	The callback function was not able to acquire the needed locks on other variables. <*lock cmdID*> identifies the current lock holder.
TYPE	A slice was specified but the variable type is not STRING (no callback).
UNKNOWN	The object was not found (no callback).

When accessing variables which are located on an attached subserver, additional error codes can occur (see section 6).

Every requested object in the GET command will be answered by exactly one DATA INLINE block. The blocks will occur in the exact same order as the object specifications did in the GET command.

After the completion of the command, the server generates a final status message as described in section 4.

4.3. SET — Updating variable contents

Writing to variables in done with the SET command. One SET command can be used to assign values to multiple objects:

<*cmdID*> SET <*object*>=<*value*>[,<*value*>][;<*object*>=<*value*>[;...]]

<*object*> can be specified as explained in section 4.1 but has to be a variable. It is not possible to assign values to modules or properties.

If multiple variables are addressed in one object (by using an appropriate <*index*> entry, see section 4.1.1), the values are to be given as a comma-separated list. In this case and also if multiple objects are specified then these will be processed sequentially. If parallel execution is desired, separate SET commands need to be sent.

Strings must always be properly escaped and enclosed in double quotes (see section 7.1).

The server will send an immediate acknowledge (or error) message after parsing the command as described in section 4.

Whether a variable is finally updated in the server (and the callback function gets executed) depends on the client's write permission and the value itself (e.g. whether the value matches the configured minimum and maximum values). The server will check these constraints before it calls any callback function.

After this update (and a possible callback execution), the server will send the client exactly one `DATA` block for every object in the command and in the order these objects were listed in the command. This block tells the outcome of the execution. This can be one line of the format

<*cmdID*> `DATA OK` <*object*>

if all value(s) could be written to the variable(s). If some or all values could not be written, the server will send

<*cmdID*> `DATA ERROR` <*object*> <*error*>[`,`<*error*>[`,...`]]

instead. For variables in the assignment that did not produce an error, an empty string will be sent. Therefore, the returned comma-separated list will exactly contain one element for each variable in the object.

The following tables show the errors that can occur. If the error occurs before the callback would have been called, this is indicated by "no callback".

Error	Description
`BUSY`	The callback is non-reentrant and already running (no callback).
`DENIED`	Write access is not granted (no callback).
`DIMENSION`	The index is out of bounds (no callback).
`FAILED` <*code*>	The callback function returned a fatal error (for additional information a <*code*> can be returned).
`INVALID`	A module or property was specified (no callback).
`LOCKEDBY` <*lock cmdID*>	The callback function was not able to acquire the needed locks on other variables. <*lock cmdID*> identifies the current lock holder.
`RANGE`	The value is not within the configured range (no callback).
`TYPE`	The type of the assigned value is not valid for this variable or a slice was specified and the variable type is not `STRING` (no callback).
`UNKNOWN`	The object was not found (no callback).

When accessing variables which are located on an attached subserver, additional error codes can occur (see section 6).

After the completion of the command, the server generates a final status message as described in section 4.

4.4. `ABORT` — Stop an executing command

When accessing variables with a callback function, commands may take some time to execute or might even hang (if the callback is designed that way). To abort such a running command, OpenTPL features a

<*cmdID*> `ABORT` <*running cmdID*>

command.

> **Implementation Note** The server implementation of the `ABORT` command should send a notification to the running callback requesting it to end (which of course can be ignored either on purpose or due to errors in the callback code).

If 0 is specified for <*running cmdID*> all commands of the current connection will be aborted (except the issued abortion command itself). If commands of other connections need to be aborted the extended command id must be used.

This command will be acknowledged by the server as described in section 4. Additionally to the there mentioned states, the following states can also be reported by the server:

State	Description
`ERROR DENIED`	The command that should be aborted was issued by a connection with a lower `RLEVEL` (for `GET`) or with a lower `WLEVEL` (for `SET`) than the current connection.
`ERROR NOTRUNNING`	No `GET` or `SET` command with the <*running cmdID*> is currently active.

The (successfully) aborted command will at least produce a

<*running cmdID*> `COMMAND ABORTEDBY` <*cmdID*>

message. (If the command was issued by another connection, the client issuing the `ABORT` command will not receive this message.)

After the `ABORT` command has finished the server will generate a final line as described in section 4. Here, the `ABORTEDBY` state cannot occur, as `ABORT` commands cannot be aborted. Additionally, the following state can also be reported:

State	Description
`TIMEOUT`	The running command did not react to the notification to abort within a reasonable timeout and may still be running.

5. Event reporting

If a callback function or a monitoring thread detects something unusual, the OpenTPL server will send an event message to the user:

<*cmdID*> `EVENT` <*type*> <*object*>:<*number*> [<*description*>]

If the event is raised by a callback function and therefore related to a currently executing command, <*cmdID*> will be set according to the command's id. Otherwise the special id 0 is used.

Events will always be published to all open connections of the server. On connections other than the event issuer's connection, events will be reported with an extended id.

However, by setting `SERVER.CONNECTION.EVENTMASK` to the desired bitmask, a client can limit the received events for its connection (e.g. `SERVER.CONNECTION.EVENTMASK=3` will only allow `WARN` and `ERROR` events to occur on this connection).

The following event types are currently defined

Bitmask	Type	Meaning
8	DEBUG	debugging message, should not occur in release versions
4	INFO	informational message
2	WARN	warning message
1	ERROR	error message

The <*object*> will identify the source of the event as a valid and unique (without multiple indexes etc.) OpenTPL object (see section 4.1). The <*number*> identifies the event and <*description*> can give additionally information.

> **Implementation Note** To enable a later look-up of occurred events, the server may feature a log module (see section 9.5).

6. Attached modules on subservers

OpenTPL supports the integration of modules (module arrays) stored on other OpenTPL servers as attached modules (module arrays). It is however not possible to attach individual variables or variable arrays. In the server configuration, the URL to the subserver is configured (see section B.2.1) for the module that should be attached. Clients can query if a module is local to the server or attached by checking the `!ATTACHED` property (see sections 8.2 and 8.3).

Attached modules (and their contents) should behave exactly as local modules. If there is a problem with the subserver, additional error codes can occur in the `DATA` blocks of `GET` (see section 4.2) and `SET` (see section 4.3) commands:

Error	Description
AUTHFAIL	Authentication with the subserver failed.
ABORTED	Command execution on the subserver was aborted.
CONNFAIL <*code*>	The connection to the subserver failed for unknown reasons (for additional information a <*code*> can be returned).

7. Variable data types

The following data types must be supported by OpenTPL servers:

No	Type (long)	Type (short)	Description
0	NULL		undefined type (should not occur)
1	INT	i	signed 64 bit integer
2	FLOAT	f	IEEE 64 bit floating point number
3	STRING	s	string (length limited only by memory)

The property `TYPE` can be used to determine the data type of a specific variable. Refer to section 7.1 for how unreadable characters in strings are escaped.

> **Implementation Note** The implementation of variable data types should be flexible and type checking should be weak, i.e. assigning a numerical value to a string should just convert the numerical value in a string. The inverse (assigning a string containing a numerical value to a numerical variable) should of course do just the opposite: convert the string to an integer

or floating point number. Strings need to be exchangeable unchanged when using `GET` and `SET` (proper quoting and unquoting on conversion).

7.1. Quoting strings

Strings must always be enclosed in double quotes. Beside the double quote character and the backslash, all characters from ASCII 32 to ASCII 255 are allowed. All other characters need to be quoted by using `\ooo` where `ooo` specifies the ASCII code in octal notation or `\xhh` where `hh` specifies the ASCII code in hexadecimal notation. There are some shortcuts for often used characters:

Char	Description
`\0`	ASCII 0 (NUL)
`\a`	ASCII 7 (BEL)
`\b`	ASCII 8 (BS)
`\f`	ASCII 12 (FF)
`\n`	ASCII 10 (LF)
`\r`	ASCII 13 (CR)
`\t`	ASCII 9 (HT)
`\v`	ASCII 11 (VT)
`\\`	ASCII 92 (\)
`\"`	ASCII 34 (")

8. Object properties

Properties must be accessible for every client that has successfully authenticated itself to the server.

There are a few properties that are supported by all object classes. These allow an "explorer" to completely readout the data structure of any OpenTPL server. These properties are[1]

Property	T	Description
`INDEX`	i	The internal object number, used for the "`<...>`" syntax
`CLASS`	i	Object class, see table below
`NAME`	s	Name of the object (can be used to retrieve the name of an object accessed with "`<...>`")
`INFO`	s	Additional description of the object (can be empty)

The following object classes are defined:

No	Type (long)	Type (short)	Description
1000	`NOTHING`		undefined object (should not occur)
1001	`ROOT`		the root object (contains the top level modules)
1002	`MODULE`	M	a module
1003	`MODULEARR`	M[]	an array of modules

[1] In this and the following tables, the "T" column will use the short type specification for the container type (see section 8) or for the variable data type (see section 7).

No	Type (long)	Type (short)	Description
1006	VARIABLE	see section 7	a variable
1007	VARIABLEARR	see above, with [] (i[])	an array of variables
2006	SYSVAR	see above, with ! (i!)	a variable with per-connection value
2007	SYSVARARR	see above, with ![] (i![])	an array of system variables

The following sections explain any additional properties supported by these object classes.

8.1. The ROOT object

The root object contains all top level modules of an OpenTPL server. Its properties are accessed with GET !<*property*>.

Property	T	Description
MEMBERS	i	Number of subobjects in this object (in this case number of top level modules)
OBJECTCOUNT	i	The total number of subobjects in this object, including all recursive subobjects but not itself

8.2. The MODULE object

A module object can contain other modules and variables (both can also be arrays).

Property	T	Description
ATTACHED	i	The module is local to the server (0) or on an attached subserver (1). In case, the module is part of an attached module array, this property will only be 1 if the module is attached to a subsubserver. (In this case, the ATTACHED property of the module array (see below) will however be 1.) The same holds true for submodules of the attached module. Here, the ATTACHED property is only 1, if this submodule itself is again located on a subsubserver.
MEMBERS	i	Number of subobjects in the object
OBJECTCOUNT	i	The total number of subobjects in this object, including all recursive subobjects but not itself

8.3. The MODULEARR object

All modules in a module array must have the same substructure.

Property	T	Description
ATTACHED	i	The module array is local to the server (0) or on an attached subserver (1). For the entries of the module array (and other submodules as well!), the property will not be 1, unless these submodules are again located on a subsubserver.
COUNT	i	The number of elements in this array
OBJECTCOUNT	i	The total number of subobjects in this object, including all recursive subobjects but not itself

8.4. The VARIABLE object

Property	T	Description
CALLBACK	s	Symbolic name of callback handler (or NULL, if the variables has no callback handler)
CALLBACKTYPE	i	Defines whether the callback is reentrant (2) or not (1) or not present (0)
INIT	as var.	Initial value after server startup
MAX	as var.	Highest allowed value for the variable
MIN	as var.	Lowest allowed value for the variable
RLEVEL	i	Maximum privilege level for reading from the variable
RLOCK	i	<*cmdID*> of the current read lock holder or 0 for no lock
TYPE	i	A number declaring the type of the variable, see section 7
WLEVEL	i	Maximum privilege level for writing to the variable
WLOCK	i	<*cmdID*> of the current write lock holder or 0 for no lock

Implementation Note A simple privilege system must be implemented by assigning a read and write level to every variable. The client is only allowed to read from or write to this variable if its privilege level is lower or equal. The lowest level for clients is zero, therefore a !RLEVEL or !WLEVEL of −1 indicates a read- or write-only variable (see section B.2.3).

Variables can also have a minimal and maximal value[2]. New values must be checked against these limits before they are written to the variable. If a initial value is defined, this value must be assigned to the variable during server startup. If the variable has a callback function this callback must be called during the initialization with special access mode parameters.

8.5. The VARIABLEARR object

All elements of a variable array must have the same properties since they share one property container to save memory. Variable arrays may only contain variables, not modules etc.

Property	T	Description
COUNT	i	The number of elements in this array
OBJECTCOUNT	i	The total number of subobjects in this object, including all recursive subobjects but not itself

8.6. The SYSVAR and SYSVARARR objects

This object is almost identical to a VARIABLE respectively VARIABLEARR object and is addressed the same way. There is however a subtle difference: the value of a VARIABLE is shared between all connections that are accessing the server. If connection 1 modifies the value, connection 2 will read the value that connection 1 has set to it. SYSVAR object values are per-connection based. I.e. different connections can have different values at the same time and changes apply only to their private value. The properties are identically to those of VARIABLE resp. VARIABLEARR and will carry the same values for all connections.

[2] Only for numerical types, i.e. INT and FLOAT

9. The SERVER module

There is only one special module that must exist in every OpenTPL server. It is named SERVER and contains several variables and submodules which are discussed in this chapter. All other modules can be freely named by the user.

The following objects exist in the SERVER module:[3]

Name	T	R	W	Description
LOAD	f		-1	Server load (1.0 means all executors busy, 2.0 means number of queued commands equals executors etc.)
LOAD_DETAIL	s		-1	System load as string with details
SHUTDOWN	i	-1	0	Shutdown the server on write access and return the written value as exit code
STARTTIME	f		-1	Server start time [seconds since 01.01.1970 00:00:00]
UPTIME	f		-1	Server run time [s]
VERSION	s		-1	Server version string (as given in welcome message)
CONNECTION	M			Access specific information on the current connection (see section 9.1)
CLIENT	M[]			Access information on all currently connected OpenTPL clients (see section 9.2)
ACCOUNTS	M			Access information on, change or remove existing accounts or add new accounts (see section 9.3)
INFO	M			Information about what is controlled with this server (see section 9.4)
LOG	M			Access to the event log (see section 9.5)
SYSTEM	M			Access to the system the server runs on (see section 9.6)

9.1. The SERVER.CONNECTION submodule

The client can retrieve connection specific information and set several configuration parameters. These parameters are internally saved as SYSVAR, therefore they can be different for each active connection.

Name	T	R	W	Description
ABORT_ON_DISCONNECT	i!			Configures, if all running commands should be aborted in case the connection is closed, valid values are: **0** no **1** yes

[3] In this and the following tables, the "R"/"W" columns specify the maximum read level (RLEVEL)/write level (WLEVEL) that is allowed to read/write the variable (see section B.2.3). A level of "-1" thereby means, that read/write access is not allowed at all, while an empty entry means, that there are no restrictions for reading/writing, e.g. that it is allowed by any level.

Name	T	R	W	Description
EVENTMASK	i!			Bitmask telling which event types should be passed on to the current connection: **Bit 0** ERROR **Bit 1** WARNING **Bit 2** INFO **Bit 3** DEBUG
ID	i!		-1	Connection number (as reported in the TPL2 introduction message)
ADDRESS	s!		-1	Source IP address, e.g. address the client is connecting from
USERNAME	s!		-1	Username used for authentication
PASSWORD	s!	-1		Set a new password for this user (PLAIN authentication type only)
RLEVEL	i!		-1	Currently active read level
WLEVEL	i!		-1	Currently active write level
STARTTIME	f!		-1	Time, when the connection was established [seconds since 01.01.1970 00:00:00]
UPTIME	f!		-1	Time, since the connection was established [s]
COMMAND_RATE	f!		-1	Command rate of the current connection [$1/s$]

9.2. The SERVER.CLIENT[] submodule

This module offers information on all currently connected clients. Connections can be terminated and the users can be denied for logging in again. The module array has as many members as client connection slots are available on the server.

Name	T	R	W	Description
ID	i	0	-1	Connection number (as reported in the TPL2 introduction message) of this connection
ADDRESS	s	0	-1	Source IP address, e.g. address the client is connecting from for this connection
USERNAME	s	0	-1	Username used for authentication of this connection
RLEVEL	i	0	-1	Currently active read level of this connection
WLEVEL	i	0	-1	Currently active write level of this connection
STARTTIME	f	0	-1	Time, when this connection was established [seconds since 01.01.1970 00:00:00]
UPTIME	f	0	-1	Time, since the connection was established [s]
COMMAND_RATE	f	0	-1	Command rate of this connection [$1/s$]
DROP	i	-1	0	Disconnect this client immediately by writing 1
BLOCK	i	-1	0	Block the account for future login attempts by writing 1 (see section 9.3 for detailed user account management functions, including unblocking an account)

9.3. The SERVER.ACCOUNTS submodule

This modules allows management of the available user accounts, including the creation of new accounts and deletion of existing accounts.

Name	T	R	W	Description
USERNAMES	s	0	-1	A comma separated list with the usernames of all existing accounts
BLOCKED	s	0	0	On write: Disable an account by writing the corresponding username On read: A comma separated list with the usernames of all disabled accounts
UNBLOCKED	s	0	0	On write: Enable an account by writing the corresponding username On read: A comma separated list with the usernames of all enabled accounts
REMOVE	s	-1	0	Remove an account by writing the corresponding username
MODIFY	M			Add a new account or modify existing ones

9.3.1. The SERVER.ACCOUNTS.MODIFY submodule

In this submodule, the client can update existing user accounts or add new accounts. It is however not possible to add or modify user accounts with lower read or write levels than those of the own account. The account information is internally saved as SYSVAR, therefore no conflicts can occur if two clients write to this module simultaneously.

To acquire information on or modify existing accounts, the username of the account is first written to the USERNAME variable. The server will then set all other variables to the values currently stored in the configuration.

Name	T	R	W	Description
TYPE	s!	0	-1	Authentication method of this account. Possible values are: **PLAIN** Account for PLAIN authentication (see section 3.2.1)
NAME	s!	0	0	The descriptive name of the account
USERNAME	s!	0	0	The username of the account. When writing an existing username, all other values in this module are overwritten with the configured values from this user account.
PASSWORD	s!	-1	0	The password used for PLAIN authentication
DEFAULT_RLEVEL	i!	0	0	Default read level of the account if not overridden in the authentication process
DEFAULT_WLEVEL	i!	0	0	Default write level of the account if not overridden in the authentication process
MIN_RLEVEL	i!	0	0	Minimal read level the account can acquire in the authentication process
MIN_WLEVEL	i!	0	0	Minimal write level the account can acquire in the authentication process

Name	T	R	W	Description
BLOCKED	i!	0	0	The enable state of the login account. Possible values are: **0** account is enabled **1** account is disabled
ADD	i!	-1	0	Add the account by writing 1
UPDATE	i!	-1	0	Update an existing account by writing 1
REMOVE	i!	-1	0	Remove the account by writing 1

9.4. The SERVER.INFO submodule

These fields can be filled with application specific contents, e.g. information about what is controlled with this server. This information can be then used by client programs to double-check if they are connected to the correct server.

Name	T	R	W	Description
DEVICE	s		-1	Device or application of this server
FLAGS	s		-1	Customer specific
INFO	s		-1	Customer specific
MANUFACTURER	s		-1	Customer specific
VENDOR	s		-1	Customer specific

9.5. The SERVER.LOG submodule

To enable clients to look up already occurred events (in case they connected afterwards), the server will log these events in this submodule. The format for these entries is

<*timestamp*> <*cmdID*> EVENT <*type*> <*object*>:<*number*> [<*description*>]

The <*timestamp*> is in the standard unix format (seconds since 01.01.1970, midnight). The remaining format is exactly the same as it was used during the original reporting of the event. The <*cmdID*> is always specified as extended id. The entries are separated either by CR-LF or LF and concatenated to a single string.

Name	T	R	W	Description
CLEAR	i	-1	0	Erase the entire log buffer on writing 1
COUNT	i		-1	Number of currently saved events
EVENTMASK	i		0	Bitmask telling which event types should be logged: **Bit 0** ERROR **Bit 1** WARNING **Bit 2** INFO **Bit 3** DEBUG
EVENTS	s		-1	All currently saved events in the described format

9.6. The SERVER.SYSTEM submodule

This module provides useful information about the operating system the server runs on. Furthermore, specific tasks regarding the system can be performed. Depending on the privilege level of the server process and the implementation not all functions may be fully supported.

Name	T	R	W	Description
ARCHITECTURE	s		-1	System architecture
CPUS	i		-1	Number of system CPUs
HOSTNAME	s		-1	Hostname
LOAD	f		-1	System load (1.0 means one CPU is fully busy)
OSTYPE	s		-1	Operating system name (e.g., Linux)
OSVERSION	s		-1	Operation system version (e.g., 3.5.0 for Linux)
REBOOT	i	-1	0	System will be rebooted on writing 1 (will require the server to run with sufficient privileges)
SHUTDOWN	i	-1	0	System will be shutdown on writing 1 (will require the server to run with sufficient privileges)
STARTTIME	f		-1	System start time [seconds since 01.01.1970 00:00:00]
UPTIME	f		-1	System uptime [s]

A. Additional implementation notes and requirements

A.1. Configuration files

The configuration of the server should be definable with ASCII configuration files. The content of the server (i.e. the variables and modules structure) must be definable with an ASCII OpenTPL DDF (Data Definition File). The OpenTPL standard requires every implementation of a server to understand the OpenTPL data definition as described in section B.

A.2. Callbacks

Upon access to variables the server should perform certain actions that are associated with the variable name. E.g. getting a variable MOTOR.TEMPERATURE should read out some sensor and return the current sensor value. The server must also be able to call functions that provide initial, minimum and maximum (and for array objects also dimension) values. The OpenTPL DDF features an object data field that can contain a symbolic name for a function that should be called on access of that variable. The server must therefore be able to execute functions by a symbolic name. A possible implementation could build the server into a library and provide it with the possibility to register callback functions with a symbolic name in it. The server program would define these callback functions and would be linked against the server library. The library then would call the program's callback functions when it loads the server configuration and for every GET / SET operation.

B. OpenTPL DDF

The OpenTPL DDF describes the entire module and variable layout of the server and also defines variable types and properties (including a symbolic callback handler name). Additionally all error

messages are defined here. The format of this file is part of the OpenTPL standard to ensure independence of the data of the actual OpenTPL implementation. The `SYSTEM` module should usually be in an own file and should only be changed carefully. Depending on the implementation this may also be fixed in the server itself.

B.1. Format of the DDF

- Each OpenTPL DDF (OpenTPL Data Definition File) **must** begin with a line containing only the string `TPL2`.
- The hash mark character `#` serves as a comment start token. Everything in a line following the hash mark must be ignored.
- The DDF contains one or more sections. A section is a line that contains only a `[`<*section-identifier*>`]` string. This section identifier should only contain alphanumerical characters.
- At least one section is required to have the section-identifier `TPL2Sys@ROOT`. This section contains all user defined top-level module structures and variables.
- Every defined structure has a section of its own which describes the structure's variables and substructures.
- A section itself contains at least one entry, that is a line containing only a string in the format <*container-identifier*> `= { ... }` which will be discussed below. Every entry specifies a data container (i.e. a `MODULE` or a `VARIABLE` or an array of these objects).
- A `MODULE` container can specify an event text section. This is a special section whose container-identifiers must be equal to the event number associated with that event text.
- Optional a section with localized event descriptions can follow.

B.2. Format of a data container

A data container is defined by a line containing only

<*container-identifier*> `= {`<*name*>`,` <*array*>`,` <*class*>[`,` <*argument*>[`, ...`]]`}`

<*container-identifier*> is an unique id that identifies the object. It must not contain characters except [a-z, A-Z, 0-9]. If the entry defines a substructure (see class arguments below), this substructure will be filled with entries of a section that is named `[`<*container-identifier*>`]`.

<*name*> is the symbolic name by which the object is later addressable.

<*array*> is 0 (if the object is no array) or an integer specifying the dimension of an array of objects. The array index is counted always 0...(dimension-1). Instead of a number also the special `NULL` value can be specified. This implies that during the creation of that container a callback function should be called. This callback function must return the array dimension. This allows dynamical array dimensions depending on the callback result.

<*class*> is the container's class, which can be one of the following: `MODULE`, `VARIABLE` and `SYSVAR`. The first class can be used to define data structures, i.e. containers containing other containers.

Depending on the class there are a number of arguments to that container class. Since the comma (`,`) character is used as delimiter for the fields of a section entry, fields containing text values should be quoted (see section 7.1).

B.2.1. Arguments of a MODULE container

<*container-identifier*> = {<*name*>, <*array*>, <*class*>, <*is-attached*>, <*connect*>, <*callback*>, <*info*>}

<*is-attached*> can be either 0 if the module is local in the server or 1 if it is a linked-in subserver (see section 6).

<*connect*> if <*is-attached*> set to 1, this specifies the subserver's connection details. It must be given as a string `TPL2://[user[:password]@]hostname:port[/PATH.ON.SUBSERVER]`. By using the optional path on the subserver, it is possible to attach also modules and submodules rather than an entire subserver.

<*callback*> is the symbolic name of a callback handler for that module (see section B.2.2 for more details). A callback handler for any container can be used for dynamic array dimensions at server start time if <*array*> is `NULL`.

<*info*> is a free text to explain the meaning of that module. The server must initialize the `!INFO` property with that text.

B.2.2. Arguments of a VARIABLE container

<*container-identifier*> = {<*name*>, <*array*>, <*class*>, <*type*>, <*rlevel*>, <*wlevel*>, <*init*>, <*min*>, <*max*>, <*callback*>, <*info*>}

<*type*> is the data type of the variable: allowed values are `INT`, `STRING` or `FLOAT` as described in section 7.

<*rlevel*> when accessing this variable the user must have a less or equal read level to be granted access (see section B.2.3).

<*wlevel*> when accessing this variable the user must have a less or equal write level to be granted access (see section B.2.3).

<*init*> initial value of the variable. The server must initialize the variable with that value. If `NULL` is specified here and a callback handler is given, the server must call the handler and take the return value of the callback (which can again be `NULL`) as initial value.

<*min*> is the minimum value constraint for the variable. If given, boundary checking must be performed by the server before setting a value. Use `NULL` if no minimum constraint is needed. In this case and if a callback handler is given, the server must call the handler and take the return value of the callback (which can again be `NULL`) as minimum value.

<*max*> is the maximum value constraint for the variable. If given, boundary checking must be performed by the server before setting a value. Use `NULL` if no maximum constraint is needed. In this case and if a callback handler is given, the server must call the handler and take the return value of the callback (which can again be `NULL`) as maximum value.

<*callback*> is the symbolic name for the callback handler function. With `@`, an automatically created symbol based on the variable path is used: `TPL2CB_` + the full hierarchy of the variable, separated by `_`, with array indexes directly appended to the names, e.g., `TPL2CB_EXAMPLE1_TEST` for the variable `EXAMPLE[1].TEST`. However, if the container itself is an array, its index is not appended, e.g. `TPL2CB_EXAMPLE2_OTHER_TEST` for `EXAMPLE[2].OTHER_TEST[]`, which means that all array elements have the same callback handler. Callback functions will also get called during server startup if the array dimension, initial, minimum and/or maximum values are set to `NULL` (see above). In this case the callback function can provide values for these fields (or return `NULL` itself).

<info> is a free text to explain the meaning of that variable. The server must initialize the `!INFO` property with that text.

B.2.3. Permissions

On every access to a variable's value both `RLEVEL` and `WLEVEL` are checked against the user's levels. Access permission is granted only if the user's level is less or equal the variable's level. Levels range from 0 to $2^{31} - 1$. To define read-only variables, set write level to -1 (which no user ever can have). For write-only variables set read level to -1. By omitting a level in the DDF, the maximum level (0) applies (resulting in access rights for all authorized users).

The read and write levels also apply to modules configured as being located on attached subserver. However in this case, the read and write levels of the specified account name in the *<connect>* string may additionally restrict access.

B.2.4. Substitution

Every field in an entry can contain special variables that are substituted when the DDF is loaded. The following tokens must be recognized and replaced by the server on loading of the DDF:

Token	Substituted with
%i	array index of the current container. Expanded to 0 if object defines no array.
%d	Id of the container
%n	Name of the container
%p	Name of the parent container

B.3. Event descriptions

Since events and errors are dependent of the device that is actual controlled by the OpenTPL server, event descriptions should be provided by the actual application. OpenTPL servers should however support localization support and therefore the optional section `[Events_XXX]` must be used, where `XXX` is the country's telephone code (e.g. `Events_49` for Germany). This section must contain only entries of the following format:

<number> `=` "*<Description message>*"

<number> is the event or error number. The OpenTPL Server must try to find a localized message if an event occurs. It must use a default (not localized) message if none is found.

B.4. Example OpenTPL DDF

```
TPL2
[TPL2Sys@ROOT]
Test={"Test", 2, MODULE, 0, "", , "Testmodul %i"}

[Test]
Var1={"Var1", 0, VARIABLE, INT, 0, 0, 100, 0, NULL, @, "Variable in %p"}
Temp={"Temp", 5, VARIABLE, FLOAT, 1, 0, 0, -273.15, NULL, @, "Tempature %i"}
Rect={"Pair", 0, MODULE, , "Just like C++ std::pair :-)"}
```

```
[Rect]
X={"First", 0, VARIABLE, FLOAT, 0, 0, 0, NULL, NULL, , "First Entry"}
Y={"Second", 0, VARIABLE, INT, 0, 0, 0, NULL, NULL, , "Second Entry"}

#localized messages for 049 - germany
[Events_49]
0 = "Das ist ein Test"
```

This example DDF will yield the following server structure:

```
(root)
|
+-- Test
     |
     +-- Var1 (INT-VARIABLE)
     |
     +-- Temp (VARIABLEARRAY)
     |     |
     |     +-- [0] (FLOAT-VARIABLE)
     |     |
     |     +-- [1] (FLOAT-VARIABLE)
     |     |
     |     +-- [2] (FLOAT-VARIABLE)
     |     |
     |     +-- [3] (FLOAT-VARIABLE)
     |     |
     |     +-- [4] (FLOAT-VARIABLE)
     |
     +-- Pair (MODULE)
           |
           +-- First (FLOAT-VARIABLE)
           |
           +-- Second (INT-VARIABLE)
```

C. Sample communication

The following example shows a short communication from connection setup to disconnection. In this example, the data the client receives from the server is marked with "⇐" and the data the client sends to the server is marked with "⇒". (Even though the command id could be reused once the command has been completed, this is not done here for clarity.)

Dir	Line	Description
⇐	`TPL2 2.0-pl0 CONN 3 AUTH PLAIN,→ CERT ENC TLS MESSAGE Welcome`	Greeting message of server, allowing authentication and encryption
⇒	`AUTH PLAIN "dummy" "secret"`	Authentication using clear text password method with user dummy and password secret
⇐	`AUTH OK 3 4`	User dummy logged in with read-/write level of 3/4
⇒	`101 SET SERVER.LOG.CLEAR=1;→ AXIS[0,1].POS=12,15`	Set two objects, one specifies two variables
⇐	`101 COMMAND OK`	The server accepted the command
⇐	`101 DATA OK SERVER.LOG.CLEAR`	First variable was updated

Dir	Line	Description
⇐	`101 EVENT WARN AXIS[1]:142→ "Speed warn:  23"`	Warning 142 occurred in module `AXIS[1]` with specific information
⇐	`101 DATA OK AXIS[0,1].POS`	Assignment was still OK (this line could be delayed until the position is reached)
⇐	`101 COMMAND COMPLETE`	The server finished executing the command
⇒	`102 GET AXIS[0-1].STATUS;SERVER.UPTIME`	Get two objects, the first specifies two variables
⇐	`102 COMMAND OK`	The server accepted the command
⇐	`102 DATA INLINE AXIS[0-1].STATUS=0,1`	The two results for the first object
⇐	`102 DATA INLINE→ SERVER.UPTIME=123192.751`	And the result for the second object
⇐	`102 COMMAND COMPLETE`	The server finished executing the command
⇒	`103 SET AXIS[0-1].STATUS=0,0`	Set two variables in one object
⇐	`103 COMMAND OK`	The server accepted the command
⇐	`103 DATA ERROR AXIS[0-1].STATUS→ FAILED 15,FAILED 15`	Both assignments failed because the callback had an error 15
⇐	`103 COMMAND COMPLETE`	The server finished executing the command
⇒	`104 SET AXIS[0-1].SELFTEST=1,2`	Assign to two variables (this operation is assumed to take a while)
⇐	`104 COMMAND OK`	The server accepted the command
⇒	`104 GET SERVER.LOG.EVENTS`	Use the ID of a running command
⇐	`0 COMMAND IDBUSY 104`	The server complains that the ID is busy
⇐	`0 COMMAND FAILED`	The command failed to execute
⇒	`105 ABORT 104`	Abort the running assignment
⇐	`105 COMMAND OK`	The server accepted the command
⇐	`104 COMMAND ABORTEDBY 105`	Command was aborted (in this case no `DATA` lines were sent, this can vary depending when the abort happens)
⇐	`105 COMMAND COMPLETE`	The server finished executing the command
⇒	`106 BADCOMMAND`	An illegal command is sent
⇐	`106 COMMAND ERROR UNKNOWN→ [unknown command BADCOMMAND]`	The server does not understand the command
⇐	`106 COMMAND FAILED`	The command failed
⇒	`DISCONNECT`	Ask the server to terminate the connection
⇐	`DISCONNECT OK`	The server acknowledges the command and then immediately closes the connection