



4π systeme
Gesellschaft für Astronomie und Informationstechnologie mbH

TPL2

Transfer Protocol Language, V2

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

Version 1.5

Document Id:

4PI-DOC-03-008-01-1

Michael Ruder, Daniel Plasa

Revision 733 as of 5th February 2004 (michael)

Revision History:

02.12.2002	Initial Version (dp)
19.02.2003	First TPL2 version (mr)
25.02.2003	Second TPL2 version (mr)
05.11.2003	Some additions (mr)
02.02.2004	Revision of DATA syntax, removed RSA auth, added TLS (mr,dp)

©2002-2004 by 4π systeme, all rights reserved.

\$Id: TPL.tex 733 2004-02-05 00:31:21Z michael\$

Contents

1	Preface	1
2	Basics	1
3	Connection setup	2
3.1	Authentication	2
3.1.1	PLAIN authentication	3
3.2	Encryption	3
3.2.1	TLS Encryption	4
4	Server commands	4
4.1	GET — Retrieving data	4
4.2	SET — Updating variable contents	9
4.3	ABORT and KILL — Stop an executing command	12
4.4	DISCONNECT — Terminate the current connection	13
5	Event reporting	14
6	Variable data types	15
6.1	Quoting strings	16
7	Object properties	16
7.1	ROOT object	17
7.2	MODULE object	17
7.3	MODULEARR object	17
7.4	STRUCT object	18
7.5	STRUCTARR object	18
7.6	VARIABLE object	18
7.7	VARIABLEARR object	19
8	Modules	19
8.1	CONFIG — Server configuration	19
8.2	LICENSE — License handling	20
8.3	LOG — Access to logging features	20
8.4	SYSTEM — Information about underlying operating system	21

A Additional Files	21
A.1 Configuration file	21
A.2 Definition files	22
A.3 Host access file	22
A.4 User file	22
B Sample communication	22
C Implementation Guidelines for Clients	25
D Current limitations	25
E Abbreviations	26
F Glossary	26
References	26

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 TPL but as it is designed to be as flexible as possible, a lot of other applications are imaginable.

2 Basics

- 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, TPL is a human readable ASCII clear text protocol. However, binary transfer is possible as well with the binary variable type.
- All TPL commands, module names and variables are case independent. However, string variables store the data case sensitive. All server replies can be expected to be in uppercase.
- 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.
- TPL supports a simple authentication scheme based on username/password authentication.
- Most implementations will include the TLS protocol to encrypt the entire communication (making the username/password authentication safe) and to do a certificate based authentication of the communicating hosts. See section 3.2.1.
- TPL 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 or data.
- Both the server and the client can send at any time. The client must be aware that status/error messages can come in at any time!
- All functions offered by the server are mapped to variables. When the client reads from or writes to these variables, the corresponding functions are executed by the server.

- Variables are collected in functional groups — called modules — to enhance hierarchical data and function structures. There is a special module **SERVER** which contains information about the server and the system it is running on.

3 Connection setup

After opening a TCP socket connection to the (configurable) port number of the machine running the server, the server will send a greeting message of the format

```
TPL2 <PROTOCOLVERSION> CONN <NUMBER> AUTH [<METHOD>,<METHOD>,...]→  
ENC [<ALGORITHM>,<ALGORITHM>,...] [MESSAGE <MESSAGE>]
```

The protocol version is given as major.minor, e.g., “1.2”. It is possible that in some intermediate releases a patch level or similar is appended to the version. However, there will be no spaces in the version string. The connection number (between 0 and 65535) identifies the current connection unambiguously. At the moment, the only defined authentication method is **PLAIN**. If no algorithm is listed, implicit authentication is used, see section 3.1. Currently, the only encryption method is **TLS** and as of now there are no plans to add more algorithms in the future. A greeting message can be attached, e.g., telling the clients address and port number.

3.1 Authentication

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

```
AUTH <METHOD> <PARAMETERS>
```

command. Otherwise, the server will immediately send the **AUTH OK** message (see below) and the clients must not send any authentication commands.

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

If the authentication was successful, the server will reply with

```
AUTH OK <RLEVEL> <WLEVEL>
```

With these levels the client can check if it is allowed to read from or write to a variables before actually accessing it (see sections 4.1, 4.2 and 7.6).

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 server answers

```
AUTH UNSUPPORTED
```

3.1.1 PLAIN authentication

In this case the client sends

```
AUTH PLAIN <USERNAME> <PASSWORD>
```

to the server.

It is possible to disable this authentication method on unencrypted connections. In this case, the encryption protocol needs to be started before this command is given (see section 3.2.1).

3.2 Encryption

While the encryption of a connection can be started at any time, it is recommended for maximum security to start it before doing authentication, especially when using the PLAIN method.

To switch into encrypted mode, the command

```
ENC <METHOD> <PARAMETERS>
```

is used. Note, that there is no command id, even if the command is given after authentication. <PARAMETERS> is not needed for all encryption methods and its meaning varies, see below.

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

```
ENC OK
```

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

In case of any syntax error the message

```
ENC ERROR
```

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

```
ENC UNSUPPORTED
```

and in both cases the connection remains unencrypted.

3.2.1 TLS Encryption

This encryption method uses the Transport Layer Security protocol (the successor of SSLv3) as described in RFC 3546. Internally, the OpenSSL library is used.

It is invoked with

```
ENC TLS
```

4 Server commands

Since the entire functionality of a TPL server is realized with variables, the command set itself is very simple and consists only of five commands: `GET` and `SET` for reading from and writing to variables and `ABORT` or `KILL` to abort one or all running commands. Furthermore a special `DISCONNECT` command exists that asks the server to close the connection immediately. All other commands that existed in TPL1 have been superseded by variables in the `SERVER` module (see section 8).

Due to the fact that any access to variables triggers callback functions which may perform time consuming tasks, the server responses may be delayed and the server is capable of executing several commands at the same time (this might be limited due to hardware restraints). To keep track of the status of all running commands, the client assigns a unique number (between 1 and 65535) to every command it is sending to the server. All server responses concerning this command are prefixed with this number. If the client tries to execute a command under the number of a still running command the server will deny the execution. The number 0 is reserved for server messages that are not related to any command, e.g., global errors. It is also used if a command without a valid command id was specified.

Internally the server keeps track of the commands using a 32-bit number, which is composed of the 16-bit connection number (see section 3) and the 16-bit command number described above. A connection number of 0 always refers to the own connection. Therefore with `ABORT` or `KILL` it is not necessary to use the 32-bit number if the aborted command was issued by the same client.

4.1 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>] [;...]]
```

`<OBJECT>` can be a module, struct or variable. Only for a variable, the `!<PROPERTY>` can be omitted for reading the value of the variable. For a list of possible properties, refer to section 7. If multiple `<OBJECT>`s are specified or if multiple values are

retrieved from one <OBJECT> then these will be processed sequentially! If parallel execution is desired, separate GET commands need to be sent. The following <OBJECT> specifications are known by the server:

Object specification	Description
<MODULE>!<PROPERTY>	Module or module array properties
<MODULE> [<INDEX>] !<PROPERTY>	Properties of a module in an array
<MODULE>.<STRUCT>!<PROPERTY>	Struct or struct array properties
<MODULE>.<STRUCT> [<INDEX>] !<PROPERTY>	Properties of a struct in an array
<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>] {<BEG>:<END>}	Slice of a binary variable

Every <MODULE>, <STRUCT> or <VARIABLE> can either be accessed by its symbolic name (e.g., SERVER) or its number with “<...>” (e.g., <0>). The number is generated by the server (and can change between sessions) and can be retrieved by reading the !INDEX property. Additionally the !MEMBERS property can be used to determine the number of sub-objects in the current object.

[<INDEX>] can be specified in various ways to access multiple elements at one time:

Index specification	Description
[<NUM>]	single element
[<NUM>-<NUM>]	range of elements
[<NUM>, <NUM>]	several elements
[<NUM>, <NUM>-<NUM>, <NUM>]	both types mixed (any mixture is possible)
[]	all elements

(Currently, this is only possible for one sub-object per object specification. For all other sub-objects <INDEX> has to be a single number. E.g., MODULE [3,4,5].VAR [3-5] is not allowed.)

For binary and string variables it is possible to access only a specific byte range instead of reading the whole variable using “{...:...” (see above). The returned slice includes both limits.

The server sends an immediate acknowledge (or error) of the form

```
<CMDID> COMMAND <STATE>
```

The following states are currently defined:

State	Description
OK	The command is valid and will now be executed
ERROR IDBUSY	The <CMDID> is in use by a currently executing command or invalid
ERROR TOOMANY	There are too many incomplete commands (i.e. commands with some DATA blocks are still missing).
ERROR SYNTAX	Command syntax error
ERROR UNKNOWN	Unknown command

If the variable has a callback function assigned to it, this function will be executed (in case of an array once for every accessed element, in case of a slice for the entire binary or string variable) before the variable contents is sent to the client.

Depending on the variable contents, the server will return the data in different formats. Additionally, to avoid blocking the connection in case of longer requests, the amount of data sent by server at one time will be limited to a configurable length (see section 8.1).

For single requests, non-binary variables are returned with

```
<CMDID> DATA INLINE <OBJECT>=<VALUE>
```

<VALUE> is always returned as text, strings are enclosed in double quotes, special characters are escaped (see section 6.1). For <OBJECT> the exact same syntax as in the request is used, i.e. if access was by the “<...>” syntax, the reply will also use this syntax instead of names. To indicate the return of a non-initialized variable that has no valid value the special keyword NULL will be returned.

Requests which return multiple values use a syntax with less overhead:

```
<CMDID> DATA TEXT <LINES> [<OBJECTPREFIX>]
```

A total of <LINES> additional lines of the format

```
<OBJECTPOSTFIX>=<VALUE>
```

or (if this element generated an error)

```
<OBJECTPOSTFIX> ERROR <ERROR>
```

will follow. <OBJECTPREFIX> may contain a fixed part (e.g., <MODULE>.<VARIABLE>) and <OBJECTPOSTFIX> contains the running part (e.g., [<NUMBER>]) (both concatenated would yield the complete object description as given in the original command). The following errors can occur:

Error	Description
DENIED	Read access is not granted (therefore no callback function was executed)
DIMENSION	The index is out of bounds (this will only be reported for the first element with an out-of-bound index)
FAILED	The callback function returned a fatal error
BUSY	The callback function is already running and is not reentrant
LOCKEDBY <CMDID>	The callback function was not able to acquire the needed locks on other variables. <CMDID> identifies the current lock holder. (This can be a 32-bit number if the lock is held by a command issued from another connection, or a 16-bit number if it was issued from the own connection.)

For binary variables the syntax

```
<CMDID> DATA BINARY <BYTES> <OBJECT>
```

is used. Following this line, <BYTES> bytes of raw binary data will be sent.

In case of longer replies (e.g. when getting large sections of an array at once), the server will use a multipart syntax to avoid blocking the connection. This is done by inserting MULTIPART directly after DATA:

```
<CMDID> DATA MULTIPART TEXT <LINES> [<OBJECTPREFIX>]
```

for DATA TEXT blocks or in case of DATA BINARY blocks

```
<CMDID> DATA MULTIPART BINARY <BYTES> <OBJECT>
```

In this case <LINES> resp. <BYTES> is the length of the current segment. To announce that the last part has been transferred the server sends

```
<CMDID> DATA MULTIPART DONE [<OBJECTPREFIX>]
```

or in case of a binary variable

```
<CMDID> DATA MULTIPART DONE <OBJECT>
```

If due to errors no data at all could be returned from an object, the following block is sent

```
<CMDID> DATA ERROR <OBJECT> <ERROR>
```

where <ERROR> can be one of the following

Error	Description
UNKNOWN	Variable or property unknown
INVALID	A module or struct was specified (without a property)
DENIED	Read access is not granted (therefore no callback function was executed)
DIMENSION	Either the variable is no array or the index is out of bounds
FAILED	The callback function returned a fatal error
BUSY	The callback function is already running and is not reentrant
LOCKEDBY <CMDID>	The callback function was not able to acquire the needed locks on other variables. <CMDID> identifies the current lock holder. (This can be a 32-bit number if the lock is held by a command issued from another connection, or a 16-bit number if it was issued from the clients connection.)

Every requested object in the `GET` command will be answered by exactly one block in the response. Either by a `<...>=<VALUE>` or by a `DATA ERROR` as given above.

After the completion of the command, the server generates a final status message, stating whether the execution was successful or not. This message is, like the immediate acknowledge message, always sent. It has the format

`<CMDID> COMMAND <STATE>`

The following states are currently defined:

State	Description
COMPLETE	The command was executed (even so there might have been some errors)
FAILED	The command could not be executed at all
ABORTEDBY <ABORTCMDID>	The command has been aborted using the <code>ABORT</code> or <code>KILL</code> command. <ABORTCMDID> is the one of the <code>ABORT</code> command and can be a 32-bit number if this was issued from another connection or a 16-bit number if it was issued from the clients connection. If the command was aborted during disconnection (see section 4.4), the lower 16 bits of the <CMDID> will be zero.

After this message was sent, the <CMDID> is no longer in use and the server will generate no more messages with this number.

4.2 SET — Updating variable contents

Writing to variables is done with the SET command. For inline assignments, the following syntax can be used

```
<CMDID> SET <OBJECT>=<VALUE>[;<OBJECT>=<VALUE>[;...]]
```

<OBJECT> can be specified as explained above but has to be a variable. If multiple objects are addressed (by using an appropriate <INDEX> entry, see above), the values are to be given with “{... , ...}”. Strings must be enclosed in double quotes (see also section 6.1). In case of struct arrays, it is possible to use “{{... , ...}, {... , ...}, ...}”. In this case, the order of the sub-variables needs to be known (e.g., by using the <...>!NAME property). It is not possible to assign values to modules, structs or properties. If multiple <OBJECT>s are specified or if multiple values are written to one <OBJECT> then these will be processed sequentially! If parallel execution is desired, separate SET commands need to be sent.

For the assignment of binary values and to have less overhead for multiple assignments, it is possible to omit <VALUE>:

```
<CMDID> SET <OBJECT>[;<OBJECT>[;...]]
```

In this case the server expects one DATA block for every listed <OBJECT>. These blocks have the same syntax as the ones the servers returns in the GET command (section 4.1). For non-binary assignments this would be

```
<CMDID> DATA TEXT <LINES> [<OBJECTPREFIX>]
```

A total of <LINES> additional lines of the format

```
<OBJECTPOSTFIX>=<VALUE>
```

is expected afterwards. For binary variables the syntax

```
<CMDID> DATA BINARY <BYTES> <OBJECT>
```

is used. Following this line, <BYTES> bytes of raw binary data will be expected. The client is encouraged to use the MULTIPART syntax for transferring larger structures. This is done by inserting the keyword MULTIPART immediately after DATA:

```
<CMDID> DATA MULTIPART TEXT <LINES> <OBJECT>
```

for DATA TEXT blocks or in case of DATA BINARY blocks

```
<CMDID> DATA MULTIPART BINARY <BYTES> <OBJECT>
```

In this case <LINES> resp. <BYTES> is the length of the current segment. To announce that the last part has been transferred, the line

```
<CMDID> DATA MULTIPART DONE <OBJECT>
```

has to be sent. Even though it is possible to mix MULTIPART blocks from different assignments this should be avoided in case of a server with limited memory resources. After the server received the last data block (or after an error or a configurable timeout) the server sends the usual acknowledge line (see section 4.1):

```
<CMDID> COMMAND <STATE>
```

The following states are defined in this case:

State	Description
OK	The command is valid and will now be executed
ERROR IDBUSY	The <CMDID> is in use by a currently executing command or invalid
ERROR SYNTAX	Command syntax error
ERROR UNKNOWN	Unknown command
ERROR TIMEOUT	There have been too long pauses between data blocks etc.
ERROR OVERRUN	The request is too complex (e.g., it uses too much memory due to mixed MULTIPART blocks).
ERROR TOOMANY	There are too many incomplete commands (i.e. commands with some DATA blocks are still missing).

Whether the variable gets updated (and therefore the callback function executed) depends whether the client has write permission, whether the value is within the specified minimum and maximum values and whether the hardware is able to comply to the new value at that time.

Afterwards, the client will receive exactly one DATA block for every object in the command, telling 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 there was a general problem and no values at all could be written, the server sends

```
<CMDID> DATA ERROR <OBJECT> <ERROR>
```

where <ERROR> can be one of the following

State	Description
UNKNOWN	Variable or property unknown
INVALID	Module, property or struct specified
DENIED	Write access has not been granted
TYPE	The type of the assigned value is not valid for this variable

State	Description
DIMENSION	Either the variable is no array or the index is out of bounds
RANGE	The value is not within the defined range
FAILED	The callback function returned a fatal error
LOCKEDBY <CMDID>	The callback function was not able to acquire the needed locks on other variables. <CMDID> identifies the current lock holder. (This can be a 32-bit number if the lock is held by a command issued from another connection, or a 16-bit number if it was issued from the own connection.)

If multiple value have been set in one object and some failed, these will be reported in a block of the following format:

```
<CMDID> DATA ERROR <LINES> <OBJECTPREFIX>
```

A total of <LINES> additional lines of the format

```
<OBJECTPOSTFIX> <ERROR>
```

will follow. The following errors can occur:

Error	Description
DENIED	Write access is not granted (therefore no callback function was executed)
DIMENSION	The index is out of bounds (this will only be reported for the element with the smallest out-of-bound index)
RANGE	The value is not within the defined range
FAILED	The callback function returned a fatal error
BUSY	The callback function is already running and is not reentrant

Error	Description
LOCKEDBY <CMDID>	The callback function was not able to acquire the needed locks on other variables. <CMDID> identifies the current lock holder. (This can be a 32-bit number if the lock is held by a command issued from another connection, or a 16-bit number if it was issued from the own connection.)

After sending all data blocks, the server sends the final status message to complete the request:

```
<CMDID> COMMAND <STATE>
```

The following states are currently defined:

State	Description
COMPLETE	The command was executed (even so there might have been some errors)
FAILED	The command could not be executed at all
ABORTEDBY <ABORTCMDID>	The command has been aborted using the ABORT or KILL command. <ABORTCMDID> is the command id of the aborting command and can be a 32-bit number if this was issued from another connection or a 16-bit number if it was issued from the clients connection. If the command was aborted during disconnection (see section 4.4), the lower 16 bits of the <CMDID> will be zero.

After this message was sent, the <CMDID> is no longer in use and the server will generate no more messages with this number.

4.3 ABORT and KILL — 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, TPL features a

```
<CMDID> ABORT <RUNNINGCMDID>
```

and a

```
<CMDID> KILL <RUNNINGCMDID>
```

command. The difference between ABORT and KILL is the way a command is aborted. ABORT sends 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). KILL will also send a notification to the running callback and (in case this notification is ignored) shut down the running callback anyway.

If 0 is specified for <RUNNINGCMDID> 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 full 32-bit command ID must be used. This command is acknowledged in the usual way. However there are some additional states:

State	Description
ERROR NOTRUNNING	A command with that ID is not currently active
ERROR DENIED	The command was issued by a connection with a lower RLEVEL (for GET) or with a lower WLEVEL (for SET)
OK	The command is valid and will now be executed
ERROR IDBUSY	The <CMDID> is in use by a currently executing command or invalid
ERROR SYNTAX	Command syntax error
ERROR UNKNOWN	Unknown command

The (successfully) aborted or killed command will at least produce a

<RUNNINGCMDID> COMMAND ABORTEDBY <CMDID>

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

After the [ABORT or KILL command has finished as usual a

<CMDID> COMMAND <STATE>

message is sent. The following states are currently defined:

State	Description
COMPLETE	The command was executed (even so there might have been some errors)
TIMEOUT	The running command did not respect notification to abort (only with ABORT) and the command is still running
FAILED	The command could not be executed at all

4.4 DISCONNECT — Terminate the current connection

To cleanly terminate the current connection, TPL implements a

<CMDID> DISCONNECT

command. This command will only produce the

```
<CMDID> COMMAND <STATE>
```

message and <STATE> will always be <OK>. After that, the server will close the current connection immediately.

Depending on the global setting `ABORT_DISCONNECT` (see section 8.1) all running commands of that connection might be aborted afterwards.

5 Event reporting

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

```
<CMDID> EVENT <TYPE> <NUMBER> <INFO>
```

If the event is raised by a callback function and therefore related to a currently executing command, <CMDID> will be set accordingly (this means that the full 32-bit number is used when the callback function is running on a different connection). Otherwise it is set to 0.

The following event types are currently defined

Type	Meaning
DEBUG	debugging message, should not occur in release versions
INFO	informational message
WARN	warning message
ERROR	error message

The <NUMBER> will have the following format

```
«MODULENUM»[.«MODULENUM»[.<...>]] :<EVENTNUM>
```

<EVENTNUM> is a module dependent number that has to be defined in a configuration file. The other numbers tell in which module and submodule(s) the event occurred. While <EVENTNUM> provides information about the event in general (and can be translated into a textual description using the `!EVENTTEXT[]` property of the module), a more specific information can be transmitted in the <INFO> field. This can, e.g., be used to communicate the actual pressure in case of a “pressure to high” error.

To enable a later look-up of occurred events, the server features a log module (see section 8.3).

6 Variable data types

The following data types are supported by TPL:

No	Name	Description
0	NULL	undefined type (should not occur)
1	INT	standard 8-byte long integer
2	FLOAT	standard 8-byte double float
3	STRING	string (length limited only by memory)
4	BINARY	any binary data (i.e. images etc.) (length limited only by memory)

The property `TYPE` can be used to determine the data type of a specific variable. Only binary variables are sent with `DATA BINARY`. Refer to section 6.1 for how unreadable characters in strings are escaped.

6.1 Quoting strings

Strings are always enclosed in double quotes. Beside the double quote character itself, the percent sign and the backslash, all characters from ASCII 32 to ASCII 255 are directly 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
<code>\0</code>	ASCII 0 (NUL)
<code>\a</code>	ASCII 7 (BEL)
<code>\b</code>	ASCII 8 (BS)
<code>\f</code>	ASCII 12 (FF)
<code>\n</code>	ASCII 10 (LF)
<code>\r</code>	ASCII 13 (CR)
<code>\t</code>	ASCII 9 (HT)
<code>\v</code>	ASCII 11 (VT)
<code>\\</code>	ASCII 92 (\)
<code>\"</code>	ASCII 34 (")
<code>\%</code>	ASCII 37 (%)

7 Object properties

Properties are accessible for every client that has authenticated itself successfully 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 a TPL server. These properties are

Property	Type	Description
INDEX	INT	The internal object number, used for the “<...>” syntax
CLASS	INT	Object class, see table below
NAME	STRING	Name of the object (can be used to retrieve the name of an object accessed with “<...>”)
INFO	STRING	Additional description of the object (can be empty)

The following object classes are defined:

No	Type	Description
1000	NOTHING	undefined object (should not occur)
1001	ROOT	the root object (contains the top level modules)
1002	MODULE	a module

No	Type	Description
1003	MODULEARR	an array of modules
1004	STRUCT	a struct
1005	STRUCTARR	an array of structs
1006	VARIABLE	a variable
1007	VARIABLEARR	an array of variables

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

7.1 ROOT object

The root object contains all top level modules of an TPL server. Its properties are accessed with e.g., `GET !MEMBERS`.

Property	Type	Description
MEMBERS	INT	Number of sub-objects in this object (in this case number of top level modules)

7.2 MODULE object

A module object can contain modules, structs and variables (all of which can be arrays).

Property	Type	Description
MEMBERS	INT	Number of sub-objects in the object
ATTACHED	INT	Is this module an attached sub server?
CONNECT	STRING	For attached modules the address of the subserver
EVENTTEXT []	STRING	The defined textual descriptions of the module events (if an empty string is returned, the event is not defined)

7.3 MODULEARR object

All modules in a module array must have the same sub-structure.

Property	Type	Description
COUNT	INT	The number of elements in this array

7.4 STRUCT object

A struct can only contain variables (no structs or modules). It has a better memory efficiency than modules, so it should be used for collections when only variables are needed (especially when these collections are dimensioned as larger arrays).

Property	Type	Description
MEMBERS	INT	Number of sub-objects in this object

7.5 STRUCTARR object

Property	Type	Description
COUNT	INT	The number of elements in this array

7.6 VARIABLE object

Property	Type	Description
TYPE	INT	a number declaring the type of the variable, see section 6
RLEVEL	INT	maximum privilege level for reading from the variable
WLEVEL	INT	maximum privilege level for writing to the variable
INIT	as var.	initial value after server startup
MIN	as var.	lowest allowed value for the variable
MAX	as var.	highest allowed value for the variable
CALLBACK	STRING	symbolic name of callback handler (or empty, if the variable has no callback handler)
RLOCK	INT	the command id of the current read lock holder or 0 for no lock
WLOCK	INT	the command id of the current write lock holder or 0 for no lock

A simple privilege system has been 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. If desired, variables can also have a minimal and maximal value¹. New values are checked against these limits before they are written to the variable. If a initial value is defined, this value gets assigned to the variable during server startup².

¹Only for INT and FLOAT types.

²If the variable has a callback function it gets called during the initialization.

7.7 VARIABLEARR object

All elements of a variable array must have the same properties since they share one property container to save memory.

Property	Type	Description
COUNT	INT	The number of elements in this array

8 Modules

There is only one special module that exists in every TPL server. It is named **SERVER** and contains several variables and sub-modules which are discussed in this chapter. All other modules can be freely defined by the user.

The following variables exist directly below **SERVER**:

Name	Type	Access	Description
VERSION	STRING	RO	Server version in the format x.y.z plus R for release, A for alpha, B for beta versions
UPTIME	INT	RO	Server run time in seconds
NAME	STRING	RO	Server name
INFO	STRING	RO	Optional description of the server
RESTART	INT	WO	Restarts the server on write access
SHUTDOWN	INT	WO	Shut downs the server on write access
PROTOCOLS	STRING	RO	Name of the supported protocols: maxrevision, separated by space (currently only TPL2:1.2)
PROTOCOL_USED	STRING	RW	Inquire or select the used protocol (no revision can be selected)

8.1 CONFIG — Server configuration

The client can set several connection specific configuration parameters:

Name	Type	Access	Description
ABORT_DISCONNECT	INT	RW	All running commands will be aborted in case the connection is closed (0=no, 1=yes)

Name	Type	Access	Description
BLOCKSIZE	INT	RW	The maximum size of a single data block the server should send in bytes. If more data needs to be transferred, the multipart syntax is used. The minimum value is 1024 bytes. 0 means that the server will never use multipart (not recommended)

8.2 LICENSE — License handling

License handling has not yet been implemented, therefore this list is to be considered preliminary.

Name	Type	Access	Description
TYPE	INT	RO	Type of license model. (Currently 0 as no license models are defined at this time)
HOLDER	STRING	RO	Name of the license owner
VALID_FROM	STRING	RO	Date and time the license became valid
VALID_UNTIL	STRING	RO	Date and time the license expires
VALID	INT	RO	True, if the license is currently valid

8.3 LOG — Access to logging features

The log module gives access to the last events that have occurred. These are recorded in a compact file format and can be read out conveniently line by line. Furthermore it is possible to clear some or all entries.

Name	Type	Access	Description
LOG[]	STRUCT	—	Log file as an array of structs (the number of elements can vary!)
LOG[].CLEAR	INT	WO	Erases the log entry on write access
LOG[].DATE	FLOAT	RO	Date and time the event occurred
LOG[].EVENT	STRING	RO	Event number as given in the EVENT message
LOG[].EVENTTEXT	STRING	RO	Textual description
LOG[].INFO	STRING	RO	Extra information as given in the EVENT message
CLEAR	INT	WO	Erase the entire log file on write access
MASK	INT	RW	Selects which events should be logged (1=debug, 2=info, 4=warning, 8=error)

Name	Type	Access	Description
MODE	INT	RW	Logging mode: 0=endless, 1=cycling, overwrite oldest entry
SIZE	INT	RO	Current size of log file
MAXSIZE	INT	RO	Maximum size of log file. Not used for MODE=0
FREESIZE	INT	RO	Available bytes in log file. Not used for MODE=0
LOGFIRST	INT	RO	Index of the first (oldest) log entry. Always 0 for MODE=0
LOGLAST	INT	RO	Index of the last (newest) log entry

8.4 SYSTEM — Information about underlying operating system

This module provides useful information about the operating system the server runs on. Furthermore, specific tasks regarding the system can be performed.

The list for available variables is not complete and will be enhanced further.

Name	Type	Access	Description
HOSTNAME	STRING	RO	Hostname
OSTYPE	STRING	RO	Operating system name (e.g., Linux)
OSVERSION	STRING	RO	Kernel version (e.g., 2.2.19)
UPTIME	INT	RO	System uptime in seconds
LOAD	FLOAT	RO	System load
REBOOT	INT	WO	System will be rebooted on write
SHUTDOWN	INT	WO	System will be shutdown on write

A Additional Files

A.1 Configuration file

The configuration file stores defaults for the configuration parameters as well as global configuration parameter like the port number and the path to the other files. The path of the configuration itself has to be specified when initializing the TPL2 library.

A.2 Definition files

The definition files store the entire module layout of the server, define variable properties (including a symbolic callback handler name). Additionally all error messages are defined here. These files can be changed without recompiling the server even though it only makes limited sense to do so.

Any number of definition files can be specified, the different sections are added together, so modules can easily be added by putting them in a new file.

The `SYSTEM` module is usually in an own file and should only be changed carefully, as it is not recommended to add, remove or rename key variables as this violates the TPL2 standard defined in this document. However, the `RLEVEL` and `WLEVEL` properties can be adapted for its variables.

A.3 Host access file

This files stores a list of allowed and denied hosts/networks.

A.4 User file

In this file, the valid users, their passwords, RSA keys and privilege levels are stored.

B Sample communication

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

Dir	Line	Description
\leftarrow	TPL2 1.2 CONN 3 AUTH PLAIN,RSA ENC→ CAST128 MESSAGE Welcome	Greeting message of server, allowing authentication and encryption
\Rightarrow	AUTH PLAIN "example" "password"	Authentication using cleartext password with example user
\leftarrow	AUTH OK 3 3	Example user logged in with read-/write level of 3
\Rightarrow	101 SET AXIS[0,1].POS	Set two variables (not inline)

Dir	Line	Description
⇒	101 DATA TEXT 2 AXIS	Set two axis variables
⇒	[0].POS=12	First element
⇒	[1].POS=15	Second element
⇐	101 COMMAND OK	The server accepted the command
⇐	101 EVENT WARN <2>[1]:142 "Speed: 23"	Warning 142 occurred in module 2, index 1 with specific information
⇐	101 DATA OK AXIS[0,1].POS	Assignment was ok (this line could be delayed until the position is reached)
⇐	101 COMMAND COMPLETE	The server finished executing the command
⇒	102 GET <2>!NAME	Check, what module generated the warning
⇐	102 COMMAND OK	The server accepted the command
⇐	102 DATA INLINE <2>!NAME="AXIS"	Module 2 is the Axis module, so the warning occurred there
⇐	102 COMMAND COMPLETE	The server finished executing the command
⇒	103 GET <2>!EVENTTEXT[142]	Get the text for the warning (alternatively, the client could have used GET AXIS!EVENTTEXT[142])
⇐	103 COMMAND OK	The server accepted the command
⇐	103 DATA INLINE <2>!EVENTTEXT[142]=→ "Axis did not reach full speed."	The general warning text
⇐	103 COMMAND COMPLETE	The server finished executing the command
⇒	104 GET CAMERA.IMAGE{2048:3327}	Request a slice of an image (from byte 2048 to 3327 = 1280 bytes)
⇐	104 COMMAND OK	The server accepted the command

Dir	Line	Description
←	104 DATA MULTIPART BINARY 1024→ CAMERA.IMAGE	The first 1024 bytes of binary data will follow
←	!+#3)...	The binary data will be sent as raw 8 bit data
←	104 DATA MULTIPART BINARY 256→ CAMERA.IMAGE	Another 256 bytes follow
←	K+Ad1...	The binary data will be sent as raw 8 bit data
←	104 DATA MULTIPART DONE CAMERA.IMAGE	All data has been transferred
←	@as?=...	The binary data will be sent as raw 8 bit data
←	104 COMMAND COMPLETE	The server finished executing the command
⇒	105 SET CAMERA.DELTAIMAGE	Set some offset image
⇒	105 DATA MULTIPART BINARY 1000→ CAMERA.DELTAIMAGE	Transfer a first block 1000 bytes
⇒	a09j1+...	The clients sends raw 8 bit data
⇒	105 DATA MULTIPART BINARY 500→ CAMERA.DELTAIMAGE	And the last 500 bytes
⇒	yx(7!x...	The clients sends raw 8 bit data
⇒	105 DATA MULTIPART DONE CAMERA.DELTAIMAGE	All data has been sent
⇒	xZt7!-...	The clients sends raw 8 bit data
←	105 COMMAND OK	The server accepted the command
←	105 DATA ERROR CAMERA.DELTAIMAGE DENIED	The client is not allowed to write to this variable
←	105 COMMAND COMPLETE	The server finished executing the command
⇒	106 SET AXIS[0-1].SELFTTEST={1,2}	Start a self test on both axes, one level 1, the other level 2
←	106 COMMAND OK	The server accepted the command (execution takes a while)
⇒	107 ABORT 106	Abort the selftest command

Dir	Line	Description
←	107 COMMAND OK	The server accepted the command
←	106 COMMAND ABORTEDBY 107	Command was aborted (in this case no DATA lines were sent, this varies)
←	107 COMMAND COMPLETE	The server finished executing the command
⇒	108 BADCOMMAND	An illegal command is sent
←	108 COMMAND ERROR SYNTAX	The server does not understand the command
←	108 COMMAND FAILED	The command failed
⇒	109 DISCONNECT	Ask the server to terminate the connection
←	109 COMMAND OK	The server accepted the command (after this, the connection is closed)

C Implementation Guidelines for Clients

- Configure `SERVER.CONFIG.BLOCKSIZE` to limit the maximum size of a data block the server sends and use `MULTIPART` for sending large quantities of data to stay below this block size.
- Since the server will buffer all data until all parts of a multipart requests are received, the client should not initiate another multipart transfer before the first one is complete. The server will also avoid this.
- Avoid direct use of any constant (like modules numbers etc.) as these numbers may change at any time.
- Clients should always terminate the connection cleanly using the `DISCONNECT` command (see section 4.4).

D Current limitations

Due to the development state of the TPL2 server, some functionality described in this document is not yet implemented or fully working.

- Only plaintext authentication has been implemented so far.

- TLS is currently not supported
- The binary variable type is not supported
- The slice syntax for strings and binary data types is not supported yet.
- The `SERVER` module is not fully implemented yet.

E Abbreviations

Abbreviation	Description
TPL	Transfer Protocol Language

F Glossary

Term	Description
Object	The full specification of a TPL object (possibly using indices that are specifying multiple elements of arrays).
Value	The value of a single TPL variable

References