TPL2

Transfer Protocol Language, V2

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

Protocol specification and implementation guide line

Version 2.0

Document Id:

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

Michael Ruder, Daniel Plasa

Revision 369 as of March 3, 2008 (michael)

Revision History:

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	Finalized revision (dp)
16.08.2005	Revision for assimilation of Java/C++ implementations (mr)

$Id: TPL2.tex 369 2008-03-03 12:33:14Z michael$

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 TPL2 but as it is designed to be as flexible as possible, a lot of other applications are imaginable.
The TPL2 protocol is absolutely free to use by anybody for any purpose and the authors encourage anybody interested in implementing a TPL2 server or client for her own purpose 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.

1.1 Conventions in this document

This manual uses the `typewriter font` whenever TPL2 protocol text is shown. Text that is enclosed in <...> stands as a placeholder. So `AUTH` <*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 TPL2 protocol itself, they will be typeset as `<...>` and `[...]`. Special design suggestions will be marked with **Implementation Note** at the beginning of a paragraph.

1.2 Protocol features at a glance

- 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, TPL2 is a human readable ASCII clear text protocol. However, efficient binary transfer is possible as well with the binary variable type.
- All TPL2 commands, module names and variables are case insensitive and all server replies can be expected to be in uppercase. However, string and binary 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.
- TPL2 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 2.1.1.
- TPL2 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 module `SERVER` which contains information about the server and the host it is running on.

2 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 protocol version is given as major.minor, e.g., "`2.0`". It is also possible that a patch level or similar information is appended to the version. However, there will be no spaces in the version string. The connection number (32 bits, between 0 and 4294967295) identifies the current connection unambiguously. The available authentication algorithms are discussed in section 2.2 and the encryption methods are discussed in section 2.1. A greeting message can follow after the `MESSAGE` key word, telling the clients address and port number or other information.

2.1 Encryption

The encryption has to be started directly after connection setup. Additionally it is a mandatory requirement for the usage of the `CERT` authentication (see section 2.2.2). Additionally the server may be configured to reject `PLAIN` authentication requests on unencrypted connections for security reasons.
To switch into encrypted mode, the special command

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

is used. Note, that there is no command id. <*parameter*>s are not needed for all encryption methods 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.

2.1.1 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.

2.2 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 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 connections before authentication.
If no algorithms are listed, 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 optional <*parameter*> differs and is explained in the following sections. It is also possible that some mechanisms require multiple steps.
As the last 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 3.2, 3.3 and 6.4).
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. In case of a disabled mechanism – e.g. as noted above disabled `PLAIN` authentication – the server will send

`AUTH DISABLED.`

2.2.1 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.
It is possible to disable this authentication method completely either on all or only on unencrypted connections.

2.2.2 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 registered with the server, the authentication will be successful. No further parameters are needed.

2.3 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 7.1) all running commands of that connection might be aborted afterwards.

3 Commands

For a maximum of flexibility, all functionality of a TPL2 server is realized with a hierarchical structure of objects (section 3.1). 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.

> **Implementation Note** It is left to the server implementation how many commands or callbacks can be active at the same time and if the same callback can be used in multiple instances at the same time. A server implementation can deny new commands if already too many commands are running and through the properties of a variable the behavior of a callback is queryable by the client.

To read from variables, the server offers the `GET` command (section 3.2). Accordingly, for writing to variables the `SET` command (section 3.3) is offered. Both commands can take one or more object specifications and must execute these requests sequentially. Since the execution of these commands may take a while (if the underlying callback execution takes some time), the server must accept new commands at any time, i.e. before already received commands are completed. All commands are executed in parallel. To associate the server replies with the correct command, every command has a unique id (in the sense that two commands with the same id that are issued from the same connection are not possible at the same time). This unique id has to be assigned by the client and must be a number between 1 and 4294967295. All server replies will carry this command id. General server error messages, an invalid command id (also an omitted id) will be answered by the server with the special id `0`. Replies that refer to other connections (i.e. `EVENTS`) will be reported with an extended id that is composed of the connection number and the command id: the lower 32 bits of that extended id is equivalent to the command id, the higher 32 bits represent the connection number (see section 2).

> **Implementation Note** Internally any server or client implementation should keep track of the commands using an extended id, that is composed of the connection number and the command id as described above.

Sometimes the client wants to stop the execution of a running command. The server offers the `ABORT` commands to accomplish this. This command will try to cleanly terminate the command (by asking the callback function to end) which may or may not succeed.
Other commands that existed in TPL1 have been superseded by variables in the special `SERVER` module (see section 7).

3.1 Object specification

Both the `GET` and the `SET` command work on objects and therefore need one or more object specifications as parameters.
Objects in TPL2 consist of a hierarchical path of the form <*subobject*>.<*subobject*>..., where the <*subobject*> can be a module or, if it is specified last, a variable. In the following sections, this path is abbreviated with <*object*>.

3.1.1 Arrays

Some subobjects 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
[<*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 indices can specify multiple elements. Otherwise the server will return a syntax error.

3.1.2 Slices

If the object is a binary or 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. If multiple array indices were specified (see section 3.1.1) the slice specification is applied to all these elements.

3.1.3 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 6 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.

3.1.4 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 a several modules 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*>]{<*begin*>-<*end*>}	Slice of a binary variable

3.1.5 Numerical access

Beside using symbolic names for subobject specification, it must also be possible to use numbers in angle brackets: `<`*number*`>`.

> **Implementation Note** The TPL2 server should assign these numbers to all subobjects during startup. These numbers must not change during server runtime but they can change if the server is restarted, especially if the configuration files are modified.

This access mode allows the client to search for all objects of the server using properties like `!MEMBERS` and `!NAME` (see section 6):

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

3.2 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 3.1. 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:

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

The following states are currently defined:

State	Description
`OK`	The command is valid and will now be executed.
`ERROR UNAUTHENTICATED`	The user needs to authenticate first (see section 2.2).
`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 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 queue.
`ERROR UNKNOWN`	Unknown command. In this case, *<cmdID>* may be set to 0.

If <*state*> was `OK`, the server will execute the command. The optional message can contain further information about errors etc.
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 objects, 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 beside binary 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 5.1). For <*object*> the exact same syntax as in the request is used, i.e. if access was by the "`<...>`" syntax or multiple array indices 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 values were accessed, these are returned comma-separated.
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.
For binary type variables, the server will start the data transfer with

<*cmdID*> `DATA BINARY` <*object*>:<*bytes*>[,<*bytes*>]

If a slice was specified and the end lies outside the data of the variable, the returned amount of bytes will be shorter than the specified slice length. If even the slice start lies outside the data of the variable, zero bytes will be returned. In case of multiple object specifications, a comma separated list of <*bytes*> is sent. Following this line, raw eight bit binary data with a length of the sum of all returned <*bytes*> values will be sent. In case of an error, the error text is sent instead of <*bytes*>.
The following errors can occur with `DATA`:

Error	Description
`BUSY`	The callback is non-reentrant and already running.
`DENIED`	Read access is not granted (no callback function is called).
`DIMENSION`	The index is out of bounds.
`FAILED` <*code*>	The callback function returned a fatal error (for additional information a <*code*> can be returned).

Error	Description
INVALID	A module was specified (without a property)
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 neither BINARY nor STRING.
UNKNOWN	The object is unknown.

When accessing variables which are really located on an attached subservers (see section **??**), additionally the following errors are possible

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).

Every requested object in the GET command must be answered by exactly one DATA INLINE or DATA BINARY block. The blocks must occur in the same order as the object specifications do in the GET command.
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*> [[<*message*>]]

The following states are currently defined:

State	Description
COMPLETE	The command was executed (even so there might have been some DATA errors).
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 normal id if it was issued from the client's connection.
FAILED	The command could not be executed at all due (in case of a COMMAND ERROR ...).

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.

3.3 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 above 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 3.1), 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 enclosed in double quotes (see also section 5.1).
For binary variables, the `=` is replaced by a `:` and <*value*> is the number of bytes of binary data to be assigned. If binary variables were part of the `SET` command the server will expect a number of binary bytes equal to the sum of all <*value*>s immediately after the `SET` command was sent. If the specified length of the data mismatches the slice size, the length of the binary variable will change accordingly.
After the server received the `SET` command (and, if binary data is part of the command, all bytes of binary data) the server will send an acknowledge line:

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

The following states are defined in this case:

State	Description
`OK`	The command is valid and will now be executed.
`ERROR UNAUTHENTICATED`	The user needs to authenticate first (see section 2.2).
`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 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 TOOLONG`	Too much data was sent. The server will try to receive all binary data even in this case to ensure a stable communication but will discard the request.

State	Description
ERROR TOOMANY	Too many commands are already in queue.
ERROR UNKNOWN	Unknown command. In this case, <*cmdID*> may be set to 0.

The optional message can contain further information about errors etc.
Whether the variable is finally updated in the server (and the callback function gets executed) depends on the client's write permission and the value itself (whether the value matches the specified 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, 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 some or all values could not be written, the server will send

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

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.
<*error*> can be one of the following

Error	Description
BUSY	The callback is non-reentrant and already running.
DENIED	Read access is not granted (no callback function is called).
DIMENSION	The index is out of bounds.
FAILED <*code*>	The callback function returned a fatal error (for additional information a <*code*> can be returned).
INVALID	A module or property was specified.
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 defined range.
TYPE	The type of the assigned value is not valid for this variable or a slice was specified and the variable type is neither BINARY nor STRING.
UNKNOWN	The object is unknown.

When accessing variables which are really located on an attached subservers (see section **??**), additionally the following errors are possible

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).

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

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

The following states are currently defined:

State	Description
`COMPLETE`	The command was executed (even so there might have been some `DATA` errors).
`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 normal id if it was issued from the client's connection.
`FAILED`	The command could not be executed at all due (in case of a `COMMAND ERROR ...`).

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.

3.4 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, TPL2 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:

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

The following states are possible:

State	Description
`OK`	The command is valid and will now be executed.
`ERROR UNAUTHENTICATED`	The user needs to authenticate first (see section 2.2).
`ERROR DENIED`	The command was issued by a connection with a lower `RLEVEL` (for `GET`) or with a lower `WLEVEL` (for `SET`).
`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 allowed range. In this case, <*cmdID*> is set to 0.
`ERROR NOTRUNNING`	A command with the <*running cmdID*> is not currently active or an `ABORT` command itself.
`ERROR SYNTAX`	Command syntax error. In this case, <*cmdID*> may be set to 0.
`ERROR TOOMANY`	Too many commands are already in queue.
`ERROR UNKNOWN`	Unknown command. In this case, <*cmdID*> may be set to 0.

The (successfully) aborted command must 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 as usual a

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

message must be sent. The following states are currently defined:

State	Description
`COMPLETE`	The command was executed (if and only if a `COMMAND OK` was reported earlier).

State	Description
FAILED	The command could not be executed at all due (in case of a COMMAND ERROR ...).
TIMEOUT	The running command did not react to the notification to abort within a reasonable timeout and may still be running.

4 Event reporting

If a callback function or a monitoring thread detects something unusual, the TPL2 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 then the event issuer's connection, events will be reported with an extended id.
However, by setting SERVER.CONNECTION.EVENTMASK to the sum of the type bitmasks on either the root or any module, 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 indices etc.) TPL2 object (see section 3.1). 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 7.3).

5 Variable data types

The following data types must be supported by TPL2 servers:

No	Name	Description
0	NULL	undefined type (should not occur)
1	INT	signed 64 bit integer
2	FLOAT	IEEE 64 bit floating point number
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 5.1 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. Binary data and strings should also be exchangeable when using `SET`(proper quoting and de-quoting on conversion).

5.1 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 (")

6 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 TPL2 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.
1003	MODULEARR	an array of modules.
1006	VARIABLE	a variable.
1007	VARIABLEARR	an array of variables.
2006	SYSVAR	a variable with per-connection value.
2007	SYSVARARR	an array of system variables.

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

6.1 ROOT object

The root object contains all top level modules of an TPL2 server. Its properties are accessed with GET !<PROPERTY>.

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

6.2 MODULE object

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

Property	Type	Description
ATTACHED	INT	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	INT	Number of subobjects in the object.
OBJECTCOUNT	INT	The total number of subobjects in this object, including all recursive subobjects but not itself.

6.3 MODULEARR object

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

Property	Type	Description
ATTACHED	INT	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	INT	The number of elements in this array.
OBJECTCOUNT	INT	The total number of subobjects in this object, including all recursive subobjects but not itself.

6.4 VARIABLE object

Property	Type	Description
CALLBACK	STRING	symbolic name of callback handler (or NULL, if the variables has no callback handler).
CALLBACKTYPE	INT	Defines wether 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.

Property	Type	Description
MIN	as var.	lowest allowed value for the variable.
RLEVEL	INT	maximum privilege level for reading from the variable.
RLOCK	INT	the command id of the current read lock holder or 0 for no lock.
TYPE	INT	a number declaring the type of the variable, see section 5.
WLEVEL	INT	maximum privilege level for writing to the variable.
WLOCK	INT	the command id 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.

Implementation Note Variables can also have a minimal and maximal value[1]. 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.

6.5 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.
OBJECTCOUNT	INT	The total number of subobjects in this object, including all recursive subobjects but not itself.

6.6 SYSVAR and SYSVARARR object

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.

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

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.

7 Special modules

There is only one special module that must exists in every TPL2 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 SERVER:

Name	Type	Access	Description
CONNECTION	MODULE		Access connection specific information.
INFO	MODULE		Information about the what is controlled with that server.
LOG	MODULE		Access to event log.
SYSTEM	MODULE		Access to the system the server runs on.
LOAD	STRING	RO	Information about the server load.
SHUTDOWN	INT	WO	Shutdown the server on write access and return the written value as exit code.
STARTTIME	FLOAT	RO	Server's start time in seconds since 01.01.1970 00:00.
UPTIME	FLOAT	RO	Server run time in seconds.
VERSION	STRING	RO	Server version string (as in welcome message).

7.1 CONNECTION submodule

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

Name	Type	Access	Description
ABORT_ON_DISCONNECT	INT	RW	All running commands will be aborted in case the connection is closed (0=no, 1=yes).
EVENTMASK	INT	RW	Bit mask telling which event types should be passed on to the current connection (1 ERROR, 2 WARNING, 4 INFO, 8 DEBUG).
STARTTIME	FLOAT	RO	Time, when the connection was established in seconds since 01.01.1970 00:00.

Name	Type	Access	Description
UPTIME	FLOAT	RO	Duration of the current connection in seconds.

7.2 INFO submodule

The customer can set several server information, e.g. information about the what is controlled with that server. This information can be used by application programs to double-check if they connected to the correct server.

Name	Type	Access	Description
DEVICE	STRING	RO	Device or application of this server.
FLAGS	STRING	RO	Customer specific.
INFO	STRING	RO	Customer specific.
MANUFACTURER	STRING	RO	Customer specific.
VENDOR	STRING	RO	Customer specific.

7.3 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 specific as extended id. The entries are separated either by CR-LF or LF and concatenated to a single string.

Name	Type	Access	Description
CLEAR	INT	WO	Erase the entire log buffer on writing 1.
COUNT	INT	RO	Number of currently saved events.
EVENTMASK	INT	RW	Bit mask telling which event types should be logged (1 ERROR, 2 WARNING, 4 INFO, 8 DEBUG).
EVENTS	STRING	RO	All currently saved events in the described format.

7.4 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	Type	Access	Description
ARCHITECTURE	STRING	RO	System architecture.
CPUS	INT	RO	Number of system CPUs.
HOSTNAME	STRING	RO	Hostname.
LOAD	FLOAT	RO	System load.
OSTYPE	STRING	RO	Operating system name (e.g., Linux).
OSVERSION	STRING	RO	Operation system version (e.g., 2.2.19 for linux).
REBOOT	INT	WO	System will be rebooted on writing 1.
SHUTDOWN	INT	WO	System will be shutdown on writing 1.
STARTTIME	FLOAT	RO	System start time in seconds since 01.01.1970 00:00.
UPTIME	FLOAT	RO	System uptime in seconds.

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 TPL2 DDF (Data Definition File). The TPL2 standard requires every implementation of a server to understand the TPL2 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 TPL2 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. Your 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` operations.

B TPL2 DDF

The TPL2 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 TPL2 standard to ensure independency of the data of the actual TPL2 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 TPL2 DDF (TPL2 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 [, Classargs...]}`

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 folowing: `MODULE` and `VARIABLE`. The first classe can be used to define data structures, i.e. containers containing other containers.

Depending on the class there are a numbers 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 5.1).

B.2.1 Classargs of a MODULE container

IsAttached can be either 0 if the module is local in the server or 1 if it is a linked-in subserver.

Connect if IsAttached=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 option 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, also see at the VARIABLE class description 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 should initialize the `!INFO` property with that text.

B.2.2 Classargs of a VARIABLE container

Type is the data type of the variable: choose from `INT`, `STRING`, `FLOAT` or `BINARY` as described in section 5.

Rlevel when accessing this variable the user must have less or equal read level to be granted access.

Wlevel when accessing this variable the user must have less or equal write level to be granted access.

Init initial value of the variable. The server must initialize the variable with that value. If a callback handler is given, the server must call the handler and take the return of the callback as initial value.

Min is the minimum value constraint for the variable. Use `NULL` if no minimum constraint is needed. If given, boundary checking must be performed by the server before setting a value.

Max is the maximum value constraint for the variable. Use `NULL` if no maximum constraint is needed. If given, boundary checking must be performed by the server before setting a value.

Callback is the symbolic name a callback handler function. Use `@` to use callback symbol `TPL2CB_` + the full hierarchy of the variable, separated by `_`, with array indices directly appended to the names, e.g., `TPL2CB_Test1_Temp` for the variable `TEST[1].TEMP`. Callback functions will also get called during server startup if the array dimension, intial value, minimum and/or maximum values are set to `NULL`. 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 should 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 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 applies (resulting in public access).
The read and write levels also apply to attached subservers. 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 container.
%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 TPL2 server, event descriptions should be provided by the actual application. TPL2 servers should however support localization support and therefore the optional section `Events_XXX` must be used, where `XXX` is the country's ISO code (e.g. `Events_49` for Germany). This section must contain only entries of the following format:

```
Num = "Description message"
```

Num is the event or error number. The TPL2 Server must try to find a localized message if an event occurs. It must use a default (un-localized) message if none is found.

B.4 Example TPL2 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 clients receives from the server is marked with "⇐" and the data the clients 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 cleartext 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
⇐	`101 EVENT WARN AXIS[1]:142→ "Speedwarn:  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

Dir	Line	Description
⇒	`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 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
⇐	`104 DATA BINARY → CAMERA.IMAGE{2048-3327}:1280`	The server will now send 1280 bytes of binary data
⇐	`!+#3)...`	The binary data will be sent as raw 8 bit data
⇐	`104 COMMAND COMPLETE`	The server finished executing the command
⇒	`105 SET CAMERA.DELTAIMAGE:1500`	Set 1500 bytes of some partial image
⇒	`a09jl+...`	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].SELFTEST={1,2}`	Assign to two variables (this operation is assumed to take a while)
⇐	`106 COMMAND OK`	The server accepted the command
⇒	`106 GET SERVER.LOG.EVENTS`	Use the ID of a running command
⇐	`0 COMMAND IDBUSY 106`	The server complains that the ID is busy

Dir	Line	Description
⇐	`0 COMMAND FAILED`	The command failed to execute
⇒	`107 ABORT 106`	Abort the running assignment
⇐	`107 COMMAND OK`	The server accepted the command
⇐	`106 COMMAND ABORTEDBY 107`	Command was aborted (in this case no `DATA` lines were sent, this can vary depending when the abort happens)
⇐	`107 COMMAND COMPLETE`	The server finished executing the command
⇒	`108 BADCOMMAND`	An illegal command is sent
⇐	`108 COMMAND ERROR UNKNOWN→ [unknown command BADCOMMAND]`	The server does not understand the command
⇐	`108 COMMAND FAILED`	The command failed
⇒	`DISCONNECT`	Ask the server to terminate the connection (after this, the connection is closed)
⇐	`DISCONNECT OK`	The server acknowledges the command and then immediately closes the connection

D Implementation Guidelines for Clients

- 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 sending `DISCONNECT` to the server (see section 2.3).

E Abbreviations

Abbreviation	Description
TPL2	Transfer Protocol Language, version 2

F Glossary

Term	Description
Object	The full specification of a TPL2 object (possibly using indices that are specifying multiple elements of arrays).
Value	The value of a single TPL2 variable

References