DISCOM TEST STAND COMMUNICATION

Content

Introduction

In a test stand environment, the measurement computer exchanges data with the test stand control computer. The test stand control for example provides information on the current type (test instruction), test step etc. The measurement system sends information like the evaluation result and reports.

The Discom measurement computer, or more specifically the measurement application *TasAlyser*, communicates with **text commands** and replies. Text-based communication has three major advantages compared to other interfaces like bit-parallel signals:

- **Security**: because the exchange of commands between measurement system and test stand can be monitored directly, any errors in communication are easily detected. There is no doubt about the timing or sequence of commands and missing or wrong commands are immediately uncovered.
- **Flexibility**: The text-based protocol can easily be extended to new commands. On TasAlyser side, a plug-in mechanism is used to realize special commands which are not needed in all test stands. If the need for a new command or communication arises, this can be added to an existing project without disrupting the existing and working command sequence.
- **Independent from hardware**: the usual means of exchanging the commands is either UDP protocol going over the existing network connection, or serial RS232 line, which for decades has proven its robustness. There is no extra hardware needed (for UDP) or the additional costs are low (for serial). Another option is the use of Profibus, but still the communication uses text commands. More details can be found in the Appendix of this document on page 25 *ff*.

In general, the measurement system works as a ‘slave’ system with respect to test stand control: The test stand sends commands or requests, and the measurement system answers. There is no communication initiated by the measurement system, so the test stand does not need to be aware of ‘asynchronous’ input from TasAlyser.

This document in the first part describes the standard set of commands which can be used in any project. After the command reference there are some examples on the usage in a complete test run. The standard instruction set can be augmented by ‘plug-ins’ which handle additional commands. These plug-ins have to be added to the project in the setup phase. The second part of this document describes plug-ins for various purposes.

General test run sequence

Each test run (complete test of an assembly) follows this scheme:

1. Start of test run. The test stand control sends the name of the type of assembly (the “test instruction” name) to start a new test run. This is called the “Insert”.
2. Test steps. The test run is separated into test steps, for example speed ramps. Each test step has a name. Within the test run, test steps can be tested in arbitrary sequence, can be repeated or omitted. A test step starts when the test stand control transmits the name of the test step and ends with the beginning of the next test step.
3. End of test run. This has two phases: end of all test steps and end of test run. At “end of all test steps”, the final result is fixed, at “end of test run” the measurement program stores the results and can send them to the result data base. The “end of test run” is signaled by the “Remove” command.
4. Querying of evaluation results: the test stand control can query the current evaluation result (OK / not OK) at any time. Even after “end of test run”, all results and reports for the last test run can be queried.

Please also refer to the extended explanation on page 11. Also see the diagram on page 13.

TasAlyser command protocol

The TasAlyser application processes text commands and replies in form of text messages. The software component which interprets the incoming texts is called the *Command Decoder*. A typical command consists of a keyword, followed by a colon and arguments. All commands are acknowledged; most replies consist of a single number (i.e. a digit), some have longer texts.

Example: the command "`Serial: 345A6789`" sets the serial number for the current unit under test. It is acknowledged with "`1`" (the character '1', not digital 1) if the serial number was set successfully and "`0`" if the command could not be executed. Not all commands have arguments; in these cases the colon may be omitted.

Between command (colon) and argument(s) any number of spaces is allowed. Upper/lower case writing is distinguished: the command `MEASURE: ON` is not the command `Measure: On`.

Command lines which cannot be interpreted as valid commands are acknowledged by "`?`".

The commands are transferred as text followed by end-of-line characters (`cr/lf`), and the reply is also a text line (consisting for example of the single letter "`1`") followed by end-of-line characters.

Command execution may not overlap. The test stand control has to wait for the reply to one command (and evaluate the reply) before sending the next.

Execution time of most commands is well below ½ second. Only the commands `Insert:` and `Remove:` (see list below) may take up to a few seconds for execution and acknowledgement.

Communication uses 8 bit characters in the currently active code page (regional settings).

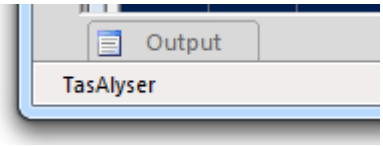
Protocol Variants

This document describes the "Handshake Standard" protocol version, which differs from "Basic Standard" protocol only in the replies to six commands.

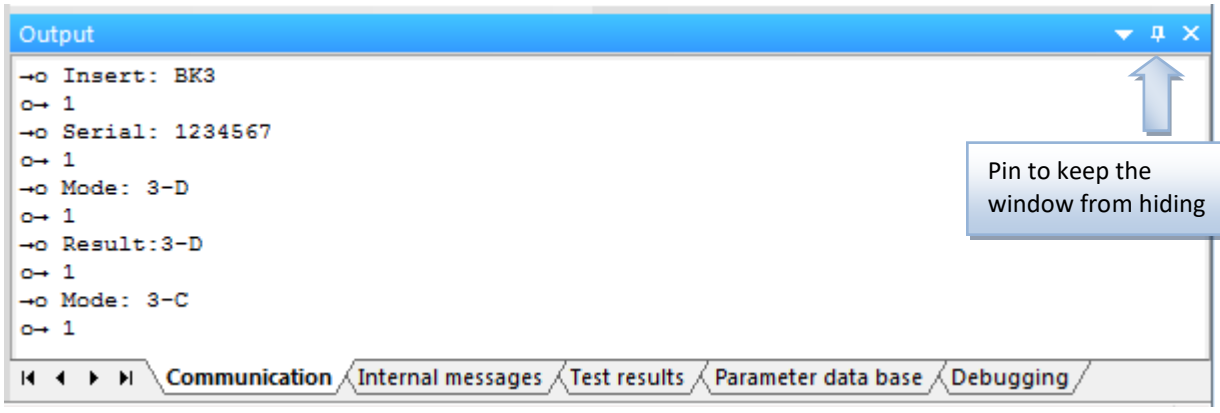
For reasons of compatibility to older test stands, as a third option the "Dpm42" protocol can be selected (see page 14). This old protocol is not subject of this document.

Observing the communication

The command based communication between test stand control and measurement system can be observed and checked in the Output window. This window is usually docked to the lower edged of the TasAlyser main window. Click on the **Output** tab to show that window. Using the 'pin' on the upper right corner of the Output window (see picture blow), you can keep it open.



In the 'Communication' tab of this window you can read the incoming and outgoing commands. Incoming commands are preceded by the symbol →○ and replies from the measurement system by ○→.



Command Execution Timing

Execution time for the various commands depends on several circumstances. Some factors are completely outside of control by the TasAlyser application, for example Windows network timeouts or parallel execution of heavy loaded tasks like RAID system rebuild. Also hard disk drives going to sleep due to energy saving settings can slow down execution of certain commands, since file access is delayed until Windows woke up the according disk drive.

Therefore, the command execution times given here are just approximations and can never be guaranteed. The best strategy is to allow for a long command timeout on test stand side, but immediately react to TasAlyser command acknowledgements.

Typical execution times (see the following section on explanation about the commands and their functions):

- Test run start (`Insert:`) can be finished in 3 seconds, but might take 10 seconds or even longer under certain circumstances (for example because the user changed settings in the parameter database). Send the `Insert` command as early in the test run as possible, thus allowing as much time as possible for execution without losing cycle time.
- Test run end (`Remove:`) also will be finished after 3 seconds in most cases but can take up to 10 seconds if hard disk file access is slow or for other reasons. Use command `EndOfTest` to split up execution time between both commands.
- Test step change (`Mode:`) needs about ½ second in most cases. Again, send the command as early as possible in the test run to allow for a longer timeout. Be aware that the `Mode` command must not be sent too early to avoid wrong triggering of measurement ramps. (For example, if the test step is triggered by a rising rpm speed starting at 300 rpm, make sure that the speed stays below that value when you send the `Mode` command until start of the actual ramp.)
- Result retrieval (`Result`, `Report`), information retrieval or inserting information (e.g. set serial number) have answering times below 500 ms. External factors have less influence on these operations, so the timing can be expected to be rather stable.

As explained above, the best strategy is to send the commands as early as possible, allowing for more timeout.

Each command gets a reply as described with the individual commands later in this manual. When the reply is sent, the command has been processed completely. The test stand control system must wait for the reply and evaluate it before sending the next command.

Timing check and Log file

As shown on the previous page, you can observe the communication records in the Output window (Communication section). Right-click within the window to call up a popup menu, then activate the command **Add time stamps** in that menu to get the timing between incoming commands and outgoing replies displayed.

The communication log from the output window is also recorded in a text file. This log file is located in the folder

`C:\Discom\Measurement\MulitRot\(`*`ProjectName`*`)\Locals\Logs\`

After the log file has reached a certain size, it gets renamed by adding a date-time string, and a new file is started. Therefore, the file without date-time in the name is the current one.

If you are having communication issues of any kind, not only timing, you should send this log file or the whole folder (compressed) to Discom.

Standard Command Set

The tables below list all commands which are understood and executed by the TasAlyser Command Decoder in standard configuration, and their replies in "Handshake Standard" protocol. If the replies in "Basic Standard" are different, these are listed, too. Additional commands are interpreted by Decoder plug-ins. Some plug-ins are described in the following sections.

When TasAlyser receives a command which is not understood (not by the standard Decoder nor by any plug-in), it replies with "`?`".

Some of the following commands need extra explanation. These commands are marked with (*) in the table. The explanation follows underneath the tables.

Test cycle

Command	Argument	Function	Reply
`Insert:`	Name of type or test instruction *Optional:* followed by ‚|' and then a serial number. Example: `Insert: ABC|12345`	Loads parameters for this type and activates the test instruction.	`Inserted` if successful, `Failed` in case of error. Possible faults: type/test instruction unknown, or previous test run not finished (`Remove` missing). *Basic standard*: 1 if successful, 0 if not.
`Serial:`	Serial number	Sets or changes the serial number information	1
`Timestamp:`	Date and time in format *yyyy mm dd hh mm ss* (*yy* in two digits is understood as 20*yy*)	Sets the time stamp for the current test run. (Default time stamp is the time of the `Insert` command.)	1 if format is valid 0 if format is incorrect
`EndOfTest:` (*)	–	End of all test steps. Sound recording is stopped.	1 if successful, 0 in case of error.
`Remove:`	–	Finishes a regular test cycle. Measurement data is stored.	`Done-`*x* when successful with *x* being the result code (see command `Result`); `Failed` in case of error (typically because `Insert` is missing). *Basic standard*: 1 at success, 0 in case of error
`Reset:`	–	Cancels the test run. No data is stored. The system is reset to initial state.	`Reset OK` *Basic standard*: 1
`Mode:` (*)	Name of test step or `$Nil`	Activates a test step Ends the current test step	`OK` at success, `Error` if not. *Basic Standard*: 1 at success, 0 in case of error
`Measure:`	‚1' or ‚`On`' ‚0' or ‚`Off`' ‚x' or ‚`Cancel`'	Starts respectively finishes the acoustic measurement	`On/Off/Cancel` at success, `Error` if unsuccessful. Basic standard: 1 at success, 0 in case of error

Test Parameters

Command	Argument	Function	Reply
`TestProcedure:` (*)	Name of a test procedure	Sets the test procedure, if needed	1 if successful, 0 in case of error.
`TestStandName:` (*)	Name for test stand	Sets the name of the test stand, if needed	1 if successful, 0 in case of error.
`TestKind:` `SetTestKind:` (*)	*n* (numerical code for test run type)	Sets the "Test Kind" (like "reference measurement" or "trial run"	1 if successful, 0 in case of error.
`SetTestProperty:` (*)	One or more test property markers	Sets the according test run properties	1 if successful, 0 in case of error.

Retrieving results

Command	Argument	Function	Reply
`Result:`	_	Produces the evaluation result (up to now)	`Result` *x* Values for result code *x*: 1 = no defects found 0 = defects occurred 2 = no evaluation performed 3 = system error *Basic standard*: only the result code *x*
`Result:`	Name of a test step	Produces the evaluation result for the given test step	Same as for the general `Result`, but referring only to the given test step.
`ClearResult:`	None or name of a test step	Clears all test results or all results for the test step	1 (even if test step name is unknown)

Retrieving reports

Command	Argument	Function	Reply
`Report:`	`Text`	Queries the measurement report (general result line + defect reports)	Text of measurement report, line by line
`Report:`	`Count`	Retrieves the number of defect codes	Number of defect codes (0 if there are no defects)
`Report:` (*)	`TextLine` *n*	Retrieves defect description for error number *n*	Description line *n* (from 1), if existing; else "–"
`Report:` (*)	`Codes`	Queries the defect codes	One defect code per line (as text). Finally always a line with "`0`" is sent to mark the end of the list.
`Report:` (*)	`CodesLine` `CodesLine` *n*	Queries the first 10 defect codes. Optional: number of digits per defect code (standard is 4)	The first 10 defect codes in one line, filled by "`0`". Example: 01230133900300000000.... contains defect codes 123, 133 and 9003.
`Report:` (*)	`CodeNo` *n*	Queries defect code no. *n*	Defect code number *n*, counted from 1. For *n* larger than the number of defects, the reply is "`0`".
`ReportDigest:`	`Format` optional `line no.`	Generates a formatted defect report	Text of defect report, line by line, followed by one extra line <end>

`ReportCodesMode:` (*)	*`Test step name`*	Queries defect codes for a single test step	Like ‘`Report: Codes`‘
`Severity:` (*)	_ Optional: test step name	Queries the level of the “severest defect”	0 = no defects found, else as specified in the parameter data base
`SeverityText:` (*)	_ Optional: test step name	Queries a text for the current severity	as specified in the parameter data base
`InstrResult:` (*)	Instrument name, optionally followed by a test step name	Produces the evaluation result for this instrument (and this test step)	1 = no defects found 0 = defects occurred

Status, special commands

Command	Argument	Function	Reply
`Status:`	_	Queries the status of the measurement system	0 = system not ready 1 = ready for “`Insert`” 2 = type/test instr. loaded
`Ping:`	(optional: arbitrary text)	Basic communication test, buffer initialization	The text which was provided as argument, or „`OK`“
`PauseWaveRec:`	1 / 0	1: pauses wave recording 0: continues wave recording	1 in most cases, 0 if not within test run
`SetComment:` (*)	Text	Sets a comment text which is stored together with the data in the measurement archive	1
`SetInfo:` (*)	Name Value	Sets an additional information “Name” to “Value”. These entries are stored in archives	0 = syntax error 1 = success
`SetComponentInfo` (*)	Component Property Value	Sets a ‘component information’ (see remarks)	0 = syntax error 1 = success
`Message:`	Text of message or “x”	Shows a message window containing the text. “x” closes the window.	1
`SetExtError:` (*)	(error code, optional values)	Adds error messages to the noise analysis results.	1 = success 0 = error
`OperatorRemove:`	(see description)	Opens a window where the operator manually closes the test run and can enter additional information	1 = success 0 = error

Explanations for some commands

Reset: On `Reset` any running test is cancelled and a test run is ended without storing any results. The wave file recording is deleted. The system returns to its initial state.

Insert: This command starts a new test run. The type name used with this command must match a type name from the parameter database. (The special type names “`$Repeat`” or “`$Again`” may be used to start a test run with the same type as before.) The Insert command is acknowledged with `Inserted` if successful and `Failed` if not. (In Basic Standard protocol, the acknowledgements are 1 and 0.) The test stand must check for the result and react accordingly!

EndOfTest: This command is optional but recommended. After `EndOfTest`, all test results are complete and the final test result is computed. Results and reports can be queried afterwards. Recording of wave data will stop at this point. If `EndOfTest` is not used, it will be called implicitly by `Remove:`.

Mode: this command is answered by `Error` (Basic Standard: 0) if no test cycle has been started (no `Insert:`) or if the test step name is unknown. The test step name “`$Nil`” ends the current test step without selecting a new one and pauses wave recording. It is not necessary to use `Mode: $Nil` before selecting a new test step.

Measure: to start an acoustic measurement, a test run has to be active (`Insert:`) and a test step has to be selected (`Mode:`). In most scenarios, the measurements are controlled by speed ramps, and you will not need to use the `Measure` command.

TestProcedure and **TestStandName:** in the parameter database, several test procedures and test stand names for the same type can be defined. (For example, there can be a short and a long test run.) The `TestProcedure:` and `TestStandName:` commands are used to select one of these. They have to be sent before the `Insert:` command and affect only the next test run.

The standard test stand name is predefined in the measurement program. If there is only one test procedure, these commands can be omitted.

TestKind and **SetTestKind**: both forms of this command are equivalent. By this command, the "test kind" or "type of test run" can be set. The test kind is stored together with all test results as a test run property. The argument for this command is a numerical code for the desired test kind. Possible values are: 1 = normal run, 2 = reference measurement, 3 = "special" measurement, 4 = test run.

The test kind set by this command is applied to the current test run and is automatically reset to "normal run" for the next test. The command can be sent before the `Insert:` and has to be sent before `Remove:`.

SetTestProperty: In addition to the "Test Kind" mentioned above, each test run can have several additional properties like "Item was returned from customer". Each property is represented by a letter in the command argument. Available properties are "R" = "item was repaired" and "D" = "item was returned from customer". So the command `SetTestProperty: R` sets the "was repaired" property. By preceding the character(s) by a minus sign (like `-R`), the properties can be removed. It is possible to set or remove multiple properties with one command.

Severity: In the parameter data base, all error codes are assigned to groups with distinct severities. This command finds the group of the "most severe defect" and sends it as answer. The reply to the command **SeverityText** is specified in the parameter data base.

Result: a test run has four possible results: not OK, OK, no evaluation or system error[1], corresponding to the numbers 0, 1, 2 and 3. If desired, the measurement system can be set up to handle "system error (3)" as normal "not OK (0)" and to handle "no evaluation (2)" as "OK (1)" or "not OK(0)", as necessary. See also page 14.

Optionally, a test step name can be given as an argument to get the result code for that specific step.

InstrResult: Using this command, evaluation results of specific „instruments" can be retrieved. For example, only the results of Crest evaluation can be queried. The first argument of this command has to be an instrument name according to the list in the parameter data base.

As a second, optional argument a test step name can be given. For example, the command "`InstrResult: Crest 3-Up`" retrieves the information if there occured a Crest defect (gear nick) in test step 3-Up.

Report: Codes / **CodesLine** / **CodeNr**: The defect codes are reported sorted by defect priority. Doubles are suppressed, that is, even if the defect with code 177 has been detected five times, only one defect code 177 is reported. Error code counting starts with 1.

Report: TextLine: same as for `Report: CodeNr` above. The returned line contains the error description text (from parameter data base) and the specification (test step, sensor etc.). Maximum line length is 120 characters. If *n* is larger than the number of errors, the reply is "—".

[1] "No evaluation" is the result, if no measurement has been done – for example, right at the beginning of a test run. "System error" signals a problem which prevents a normal test, for example a sensor defect. In this case, a OK/not OK decision cannot be made.

ReportDigest: using this command the test stand can query the elements of the defect report (error code, text, test step, …) in arbitrary order. The command syntax is

```
ReportDigest: Format
```

The 'Format' consists of a sequence of letters which describe the desired defect report elements. They are

C	E	T	M	S	V	P	D	N
error code	"external" code	text	test step	specification	value and limit	position	difference value – limit	line no.

Example: the command

```
ReportDigest: CMT
```

produces the error code, the test step name and the error text, as in

```
583 3-D Order loud
```

The elements are separated by a whitespace character. A different separator character can be given as the first character of 'Format'; any non-alphanumeric character is allowed. Example:

```
ReportDigest: |TMS
Order loud|3-D|Spectrum Intermediate shaft Sync
```

The `ReportDigest` command produces as many lines as there are defect reports, plus an extra line

```
<end>
```

If there are no defect reports at all, only the line `<end>` is produced.

Optionally, a line number can be given as a second argument. Line numbers count from 1. If a number is present, the command produces only this line of the defect report, or the line `<end>` if the number is larger than the defect count (see command Report: Count).

Ping: This command is processed directly within the decoder without any interference with the test run control. The answer is the text which is provided as argument, or "`OK`" if no argument was provided.

The `Ping` command can be used at any time to test the basic interface communication and, if needed, to get a different reply from the measurement system than the usual digits "1" or "0" (e.g. in order to initialize a reply buffer).

Important note: the `Ping` command is intentionally not locked against overlap with other commands. So if the test stand sends an `Insert` command immediately followed by a `Ping`, the reply to `Ping` may come before or after the acknowledgement to the `Insert`.

PauseWaveRec: In the usual setup, all sensor data are recorded into a wave file starting with the first test step and ending with the `EndOfTest` command. Sometimes, a test run contains sections of significant duration in between, which are of no interest for the acoustic analysis. At the beginning of such a section, wave recording can be paused with `PauseWaveRec: 1`. This will reduce the size of the wave files. At the end of the section, recording can be resumed with `PauseWaveRec: 0`, but will automatically resume with the next `Mode:` command.

SetComment, SetInfo: The information set with these commands is stored in the "Additional Information" section of the measurement archives. SetComment sets the predefined measurement comment, SetInfo an arbitrary additional information. Example: `SetInfo: MainShaftType Abc123` stores "Abc123" as value for the information "MainShaftType". `SetInfo` and `SetComment` commands have to be sent before `Remove`, and the `Reset:` command also resets all SetInfo-content.

SetComponentInfo: This command is for entering information about components of the test object, for example the serial numbers of gears inside a transmission. The command has three arguments: an 'Element Name' (like "PrimGear"), a 'Property Name' (for example "GearSerial") and a value (the serial number of the gear). `SetComponentInfo` behaves like `SetInfo` in all other respects.

SetExtError: This command used to be handled by a plug-in but is now part of the standard command set. For details read the plugin description (the plugin still exists and has extra functions) on page 16.

OperatorRemove: A dialog window opens in the measurement application where the operator must manually confirm the test run. There are three choices: “OK” (test run is valid, equivalent to Remove), “Calncel” (test run invalid, equivalent to Reset) and “Back” (test run not yet finished, return to test steps). In addition, the operator can type in additional information in that window (e.g. a comment text) which are stored together with the measurement results.

As an argument the command accepts a number which is a bit-wise OR of the options listed below. (For example, options 1 and 4 combine to argument 5, so the command is `OperatorRemove: 5`.)

Option	Function
0	Opens the window and waits for operator choice, but the `OperatorRemove` command is immediately acknowledged to test stand control
1	The operator choice “Back” is not available
2	Operator cannot change the serial number
4	The `OperatorRemove` command is acknowledged not immediately, but delayed until the operator makes a choice (and thus finishes the test run)

If no argument is given to the command, option 0 is used.

As long as the dialog window is open and the operator has not made a choice, no test run commands (`Reset`, `Remove`, `Mode`, or a new `Insert`) will be accepted from test stand control.

Comparing Handshake-Standard and Basic-Standard

The table below lists the different replies for commands in Handshake Standard protocol as compared to Basic Standard.

Command	Basic Std.	Handshake-Standard
`Reset:`	`1`	`Reset OK`
`Insert:` *`ABCD`*	`1 / 0`	`Inserted / Failed`
`Mode:` *`1-A`*	`1 / 0`	`OK / Error`
`Measure: On/Off/Cancel`	`1 / 0`	`On / Off / Cancel / Error`
`Remove:`	`1 / 0`	`Done-`*`x`*` / Failed` *`(x = Result Code)`*
`Result:`	*`x`*	`Result` *`x`* *`(x = Result Code)`*

Examples

This example shows a simple test run with test stand commands and TAS answers:

Test Stand	TAS	Description
`Reset:`	`Reset OK`	Reset to start. Cancel any possible incomplete test runs.
`Status:`	`1`	Answer must be 1 before a test run may be started.
`Insert: A17`		Type „A17“ will be tested now.
	`Inserted`	Parameters for „A17“ successfully loaded
`Serial: 4711`	`1`	Setting the serial number
`Mode: Up`		Next test step: „Up“
	`OK`	Acknowledgement: ready for „Up“
(drive a speed ramp)		
`Result: Up`		Querying the evaluation result for „Up“
	`Result 1`	Result for test step „Up”: no defects found
`Mode: Down`		Next test step is „Down“
	`OK`	(see above)
(ramp)		
(call up other test steps with `Mode` *command and execute ramps)*		
`EndOfTest:`	`1`	All test steps done. Stops sound recording.
`Result:`	`Result 1/0/2/3`	Total test result: OK / not OK / no evaluation / system error
`Remove:`	`Done-1`	Test cycle finished; overall result is OK
`Report: …`	`…`	Now, any kinds of reports can be retrieved
`Reset:`	`Reset OK`	Start next test cycle – continue as above.
`Insert: A17`	`Inserted`	...

Notes

- The system keeps all measured data, defect reports etc. until the next test run is started with `Insert`. So even after `Remove:` any kinds of reports can be queried, even multiple times.
- The command `Serial` must be sent before `Remove`. It can be sent even before `Insert` and can be repeated to change the serial number. The serial number can be any character string (only whitespace is forbidden). There is no length restriction (besides memory space).
- The parameter data base contains all valid type names. All other type names are rejected at `Insert`.
- The same is true for the test steps. Only test step names contained in the parameter data base may be used with the `Mode:` command.
- The sequence of test steps during a test run is arbitrary. Test steps may be omitted or repeated and measured in any order. If a test step is repeated by sending `Mode: X` once more, all results and defects from a previous measurement of test step `X` are deleted.
- The measurement system is in most cases set up to check speed ramps automatically and start and stop the measurement according to speed ranges. In these cases, the `Measure:` commands are not needed.
- Querying the test result for individual test steps with the `Result:` command during the test run is optional. This gives you the information if there was a problem in a specific test step.
- Always ask for the overall test result after `EndOfTest` (or `Remove`), even if you have checked the results of individual test steps during the test run. There are errors (like sensor defects)

which are not associated with specific test steps, and you would miss these errors if you only ask for the test step results.

- The reply times for all commands except `Insert:` and `Remove:` are far below 1 second. The `Insert:` command may take more time (up to 10 seconds) if changes in the parameter data base have to be updated or a new type is requested; otherwise `Insert:` takes no more than 1-2 seconds. The time required for execution of `Remove:` depends on the time taken by writing and transferring the measurement data archive but should not exceed 10 seconds. See also the remarks in "Command Execution Timing" on page 4.

Between commands and arguments and after these, any number of spaces is allowed.

Advanced Example

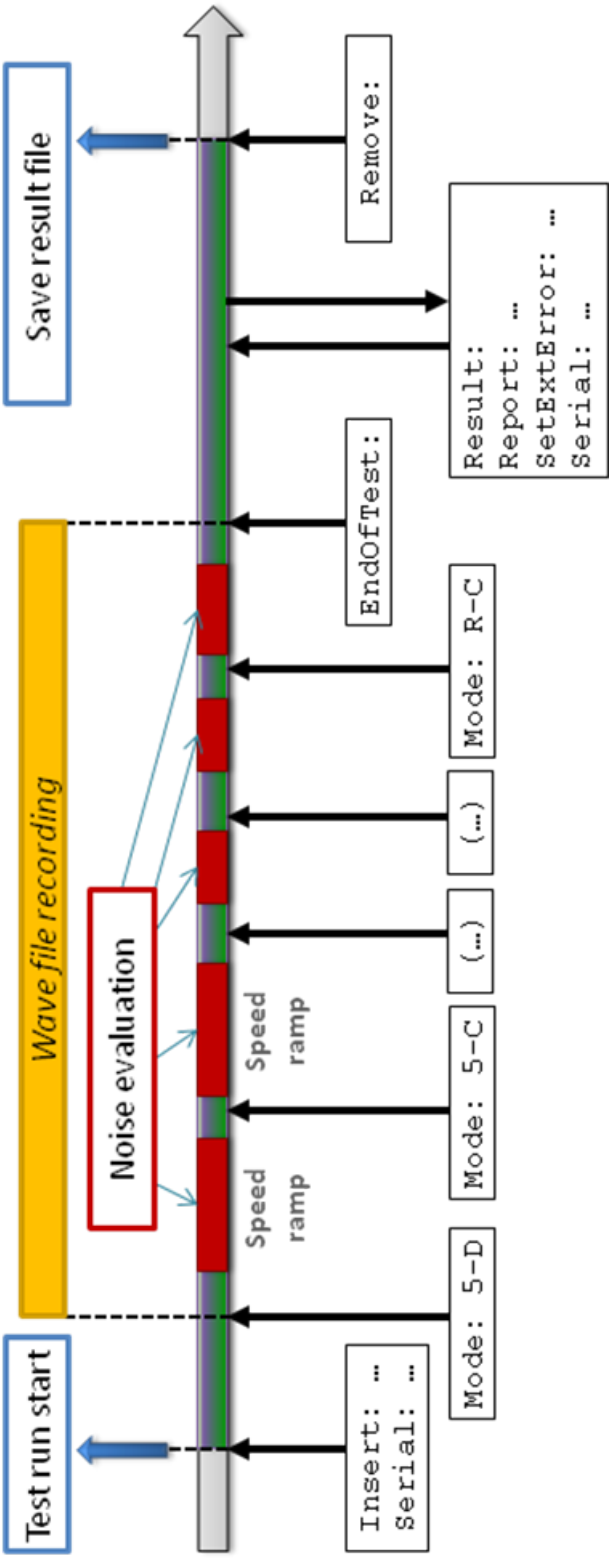
This example shows a test run with some additional, optional commands. Some of these are executed by plug-ins (see "Command Decoder Plug-ins" on page 16 *ff.*). Replies of the TAS systems are not shown; for more information about these and more details about the commands see the descriptions in the previous chapter.

`Reset:`	
`SetInfo: BoxType 5A`	Set an additional information for storing in the result file
`Serial: 1234567`	
`TestProcedure: SpecialTest`	Instead of a normal test run, a „Special" test run will follow
`Insert: PQR`	
`Ping: happy`	Is immediately answered with "happy"
`Mode: 1-D`	
(drive a speed ramp)	
`Mode: TqR`	Test step „Torque Ramp"
(drive a torque ramp)	
(call up other test steps with `Mode` *command and execute ramps)*	
`SGW: 4 5`	Start of a gear switching operation from 4 to 5*
`Mode: 5-C`	
`EGW:`	End of gear switch*
(speed ramp)	
`Result: 5`	Query result for gear 5 (assembling results from 5-D and 5-C)
`Mode: $Nil`	This command sets the wave recording to pause (until the next `Mode` command). This is recommended if there are long parts without noise analysis within the test run.
(other tests without noise analysis)	
`Mode: Steady`	In this example, "Steady" is a test step without speed ramp.
`Measure: 1`	Therefore, the measurement has to be started by a command…
(timer controlled test)	
`Measure: 0`	…and also stopped by a command.
`EndOfTest:`	
`GetValueByName: StdRMS`	Query value for the measured quantity „StdRMS"*
`SetExtError 567 33.8 25`	Add test stand error 567 to the list of acoustic errors
`Result:`	Always query the overall result!
`Serial: 987654BX856432A`	Change serial number
`Remove:`	

* These commands are handled by plug-ins; see "Command Decoder Plug-ins" on page 16 *ff.*

Test run timing

The graph below shows the timing of a typical test run:



The system records the sensor data into a wave file starting with the first `Mode` command and until `EndOfTest`.

The command `SetExtError` is an extension explained in the following section. The test stand can send error codes to the noise analysis system which are stored together with noise analysis results.

About the Result Codes

When testing a part in production, there are only two basic options: part can be sold (is OK) or cannot be sold (not OK). But actually, there are two additional situations: no test was performed, or the test could not be completed.

Accordingly, there are four result codes as answers to the `Result:` command:

Code	Meaning	Explanation
1	Test OK	No defects were found (or transmitted using `SetExtError`). The tested part can be sold.
0	Test not OK	Defects were found. The tested part should not be sold directly but repaired or recycled.
2	no evaluation	No test has been performed yet. This is the initial result at the beginning of a test run. If the system replies with 2 at the end of the test run, check whether any `Mode:` commands have been successfully transmitted.
3	System Error	A problem occurred which makes it impossible to judge whether the tested part is OK or not OK. Sensor defects or missing speed signals are typical examples for system errors. Stop testing and call an operator.

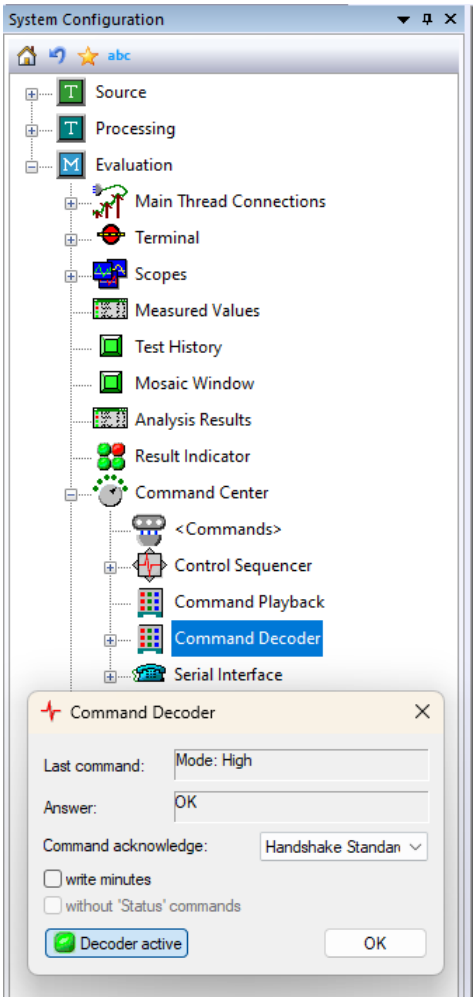
A test stand must handle all four possible results. (The reaction to a result code 3 should be typically a stop and call for an operator.)

Although it is possible in the TasAlyser measurement software to map result code 2 to 1 and result code 3 to 0, this option is reserved for use with old test stands which are not capable of handling the system error situation appropriately.

While it is possible to ask for the result for individual test steps (e.g., to stop the test run early if one test step fails), it is vitally important that the test stand control software always asks for a general `Result:` at the end of the test run (at some point after `EndOfTest`). Some errors like missing signals or internal processing errors are not assigned to a normal test step and will pass unnoticed to the test stand if the general `Result:` query is omitted.

Settings and Test Mode

To access the Command Decoder software module, go to the **System Configuration** window, unfold the **Evaluation** branch, therein open the **Command Center** node and double-click on the **Command Decoder** item:



The Command Decoder module shows in it's settings window the last received command and the reply.

If the checkbox **write minutes** is activated in the dialogue, all received commands and all sent acknowledgements are mirrored in the Output window.

The interface modules (specifically the serial interface) also offer the option of reporting the communication traffic into the Output window. The report generated by the TAS Decoder is more compact, so you may want to choose this if you only want minutes of the commands and not of the detailed communication.

Under **Command acknowledge**, the option **Dpm42 syntax** switches the replies to the commands from the Standard protocols documented in the tables above to the texts sent by the old Dpm42 Rotas measurement systems. The option "**–> [Command]**" extends the Basic Standard replies by the command which triggered the reply (e.g. "`1 [Insert]`").

You can temporarily switch off the Decoder, for example if you want to test your PLC software without having the TasAlyser react to it.

Command Decoder Plug-ins

A Decoder Plug -in is a software module which appears as a sub-module of the Tas Decoder in the system configuration. A Decoder Plug- in extends the commands and functions of the system.

Plug- ins are used in a project specific manner, so any plug-in may or may not be contained in a project.

A plug-in may also change the behavior of one or all of the standard commands described above (for example, change the replies form “1” to “<1>”) or introduce alternate names for standard commands (for example “StartTestRun” instead of “Insert”).

Sending test stand errors: “ExtError” Plugin

Via these commands, test stand control can insert additional defect messages into the defect list of the measurement program. These “external” defects are handled in the same manner as those generated by the noise analysis, i.e. they generate a “not OK” result, are stored in archives and in the measurement result database etc.

The “ExtError” Plug-In handles the command `CheckForError`, while `SetExtError` and `ExtError` are part of the standard command set and do not require the plugin.

Command	Argument	Reply

`SetExtError:`
`ExtError:`
	`Defect-code`
	`  Value Limit Position`
	`, Defect-code…`
		1: Defect message accepted
		0: not accepted, because no test running.
		2: Defect-code unknown.

`CheckForError:`	`Defect code`	1: code appears in list of current errors 0: code is not among current errors

In its basic form, the command is

```
SetExtError: 1234
```

Optionally, a measured value (floating point number) can be transmitted. A limit value and a position may be added. These elements are separated by (any number of) spaces. Example:

```
SetExtError: 1234  14.7  10.0  1200
```

Values which are not given are assumed as zero.

Please note the following points:

- Both forms of the command (SetExtError and ExtError) are equivalent.
- The `SetExtError:` command is only allowed during a test run, that is after `Insert:` and before `Remove:`.
- More than one defect code can be sent with one command. The defect codes have to be separated by commas. Examples:

  ```
  ExtError: 309  14.7  10.0  1200, 312 159.4 150.0 800
  ```

  ```
  SetExtError: 309 , 312 , 433
  ```

- The defect code is a number [2]. Each defect code must be set up in the parameter data base of the measurement project, and the details (especially the associated text) have to be specified. The choice of numbers for defect codes is completely free; they do not have to be sequential or start with 1. Take care to avoid conflicts with NVH defect codes.
- The defect code used in the command does not necessarily have to be same number as the associated defect code used in the measurement program. The association between ‘command codes’ and ‘internal codes’ is also set up in the parameter data base. But in most cases, the ‘command codes’ will exactly equal the defect codes.

[2] More precise, a positive whole number < 2^{31}

- It is not possible to send two defect messages with the same defect code (within one test run). The second `SetExtError:` with a defect code already in use will overwrite the first message. But it is possible to send any number of defects with *different* defect codes.
- By sending `SetExtError:` with a negative defect code (for example `SetExtError: -309`), the respective defect entry can be deleted.

The `SetExtError` command inserts a defect message into the result data set of the current measurement (and sets the overall result to 'not OK'). It does *not* insert a *measured value*, even if you provide a value with the command. The provided value only appears in the error information, not in the measurement results. To insert a measured value, use the **GetValues** plugin described below.

Using `CheckForError:`, test stand control can query whether a specific error code appears in the list of defect messages for the current test run. It works at any time during and after the test run.

Gear Shifting Plugin

The gear shifting plug-in handles the commands "SGW" (="**s**tart **g**ear s**w**itch") and "EGW (="**e**nd **g**ear s**w**itch"). These commands trigger modules for measurement of gear shifting processes like the gear shifting noise or the gear shifting force.

The plug-in understands the following two commands:

Command	Argument	Function	Reply
`SGW:`	`gear1 gear2`	Starts measurement of gear shifting process from gear 1 to gear 2. Example: `SGW: 2 3`	1: Measurement started 0: if the command was syntactically wrong (e.g. missing argument). 2: if a previous `SGW` has not been finished
`EGW:`		Finishes measurement of the current gear shifting process	1: no defects found (OK) 0: defects found (gear shifting not OK) 2: no SGW command was sent before.

The SGW command requires the name of two gears (gear shifting from – to). The gear names are "physical gears" (like R, 1, 2, 3…), not test step names, and have to be present in the parameter data base. Valid names for idle gear are "N" or "0" (Null). The gear names are not checked by the plug-in, so the reply to `SGW: 88 99` will be 1 even if there are no gears 88 or 99 – only the measurement modules will not produce any results.

The reply to `EGW:` is the evaluation result of the gear shift measurement. Because of the evaluation there may be a slight delay (less than ½ second) between command and reply. The defect codes for any found defects can be retrieved using the `InstrResult: 5000` command.

Example:

`SGW: 2 3`	Command: start shifting measurement for shifting from 2nd to 3rd gear
`1`	Reply: Shifting measurement started
`EGW:`	Command: shifting process finished
`0`	Reply: shifting measurement produced a not OK result

In the measurement program the option for "extended reply" can be activated. In this case, the reply to EGW consists of three digits, separated by spaces:

`EGW:` → `a b c` `(a, b, c each = 0 or 1)`

Digit *a* shows the overall result (as before), digit *b* indicates the result for the gear-out measurement and digit *c* the result for the gear-in measurement.

In the measurement application, a maximum measurement time for the gear shifting process is set (normally 10 seconds). If the `EGW` command does not follow the `SGW` within this time, the measurement is cancelled.

Ratio Test Plugin

This plug-in is necessary to trigger the ratio test and the differential test. It understands the following commands:

Command	Argument	Function	Reply
`StartRatioTest:`	_	Starts the ratio test for the current mode	1
`StartDiffTest:`	_	Starts the differential test	1
`EndRatioTest:` `EndDiffTest:`	_	Finishes a running ratio or differential test	1: no defects found (OK.) 0: defects occured (not OK)
`GetRatioValue:`	test step	Queries the measured ratio value	(measured value)
`GetInvRatioValue:`	test step	Queries the inverse ratio value	(reciprocal value)

These commands start the ratio test or differential test. The ratio test checks for the ratio of the currently active mode (gear). Both tests are finished with the `EndRatioTest:` command. The reply to `EndRatioTest:` is the ratio test result. Because of the evaluation there may be a slight delay (less than ½ second) between command and reply. If the `EndRatioTest:` command is not sent, the test ends with the next `Measue: 0` command or end of next measurement ramp.

The commands `GetRatioValue:` and `GetInvRatioValue:` query the measured ratio value for a test step. Example:

```
GetRatioValue:  3-D
4.186
GetInvRatioValue:  3-D
0.239
```

If the value is queried for a test step where no ratio has been measured, the result is 0.

DummyCommands-Plugin

The DummyCommands Plugin can modify test stand command texts and can reply to commands with predefined answers, without further actions. Using this plugin, the processing of any command can be simulated. Standard commands (like `Measure:`) can be overridden, too.

Command	Argument	Reply
`(any)`	`(irrelevant)`	as you wish

The list of recognized commands and answers are stored in the application file:

```
TasDecoderDummyCommands: {
      RESET  <R1>:RESET
}
```

In this example, the plugin catches the test stand command “RESET:” and replies with “<R1>:RESET”, while nothing else happens in the measurement program. In the list in the application file, the colon after the command and all arguments (if any) are omitted.

Please note that the commands are case sensitive. The above example catches the command “RESET:”, but not “Reset:”.

The reply to any command can be only one line of text. If the reply contains whitespace, the reply has to be enclosed in quotation marks.

The plugin can also replace commands with other commands (note the different list name):

```
TasDecoderDummyCommandReplacement: {
      RESET  Reset
}
```

This replaces the (wrong) upper-case “RESET” with the correct form “Reset”.

The plugin can also replace test step names in the Mode: command, and can override the type name given in Insert commands.

Get/Set Values Plugin

The test stand computer can retrieve measured values from the TasAlyser measurement application or set additional results. Values can be queried one by one or as a list.

To identify the measured values, each value gets a name called *ValueKey*. These names are set up in the parameter data base in the form **Alias Keys List**. Only values listed in the according data base table can be queried or set. In addition, when asking for a value, a test step has to be specified. When the test step name is omitted, the result for the current test step is delivered. The GetValues plug-in supports the commands:

Command	Argument	Reply
`GetValueByName:`	`ValueKey (test step)`	Numbers for this value.
`GetValueKeys:`	-	List of available *ValueKeys* as described below
`GetValueList:`	`(test step name)`	List of values for the *ValueKeys* as described below.
`SetValueByName:`	`ValueKey TestStep value (unit limit position)`	1 in case of success 0 if 'ValueKey' or 'TestStep' are not valid.
`SetValuesFromFile:`	`Filename`	1 in case of success; 0 in case of errors

Querying one value

To get one result at a time, the test stand uses the command GetValueByName. This is answered by a row with one or more numbers:

```
GetValueByName: RMS-Mix 3-D
2.45 2500 12.0
```

The parameter data base specifies for each *ValueKey*, how many numbers are produced. The meanings of the numbers are (in this sequence): measured value, position information, limit value, learned mean value. So in the above example, the measured value for RMS-Mix in test step 3-D is 2.45 and the limit is 12.0. The meaning of the position information depends on the value. For example, for spectral values the position is the spectral order.

If the queried *ValueKey* does not exist in the parameter database, the reply is "`undefined`". If the *ValueKey* exists, but no value has been measured in the according test step, the reply is "`n.a.`".

If the test step name is omitted (as in "`GetValueByName: RMS-Mix`"), the current test step is used.

Querying all values

To get all values for a test step, the test stand uses the command `GetValueList`. To make communication more compact, the reply does not contain the *ValueKey*-Name for each value, but a *KeyNumber*. So first the test stand has to query the association between *ValueKeys* and *KeyNumbers*. This is done with

```
GetValueKeys:
```

This command can be used at any time, even before Insert. The reply is the list of *KeyNumbers* and *ValueKeys*:

```
<ValueKeys>
number1 "Name1"
number2 "Name2"
...
</ValueKeys>
```

Example:

```
<ValueKeys>
7 "RMS-Mix"
44 "LKon_H1"
</ValueKeys>
```

The *ValueKey*-Names are always enclosed in double quotes. Only those *ValueKeys* which have not been set to 'inactive' in the parameter data base appear in the list. So, the list may be empty. The *KeyNumbers* are assigned automatically by the parameter data base, so they may not be contiguous, and the list is not sorted. The associations may changes between test runs, so it is recommended to re-query the *ValueKeys* and *KeyNumbers* before each test run.

To get all values for a test step, the test stand uses the command

```
GetValueList: (test step name)
```

If the test step name is omitted, the current test step is used. The reply is a list of *KeyNumbers* with according values (and, if set in the parameter data base, position, limit and learn data):

```
<Values (test step)>
key1 value1 (position1) (limit1) (learned1)
key2 value2 (position2) (limit2) (learned2)
...
</Values>
```

Example:

```
<Values 4-NZ>
44 75.00 98.34 88.62
7 2500.00 3.78
</Values>
```

KeyNumbers and values are separated by spaces. The list is not sorted by *KeyNumbers*, and the list may be empty.

The <Values>-List contains all values which appear in the form **Alias keys list** in the parameter data base and which *should be present* in this test step. If a value cannot exist in the current test step (like a value for the reverse gear, which only exists in reverse gear), it does not appear in the list. If the value is expected but was not measured (for example because the target speed was not reached), "`n.a.`" appears in the list:

```
<Values 4-NZ>
...
44 n.a.
...
</Values>
```

Setting a value

The test stand control can send values to the measurement system which will then be stored together with all other measurement results. The command is

```
SetValueByName: ValueKey TestStep value (unit) (limit) (position)
```

Unit, limit and position values are optional. Example:

```
SetValueByName: OilTemp Function 89.4 °C 100
```

sets a predefined value of 89.4 °C for "OilTemp" in the test step "Function" (both have to be prepared in the parameter data base in advance). The limit is 100.0, a position is not provided.

Only unit names predefined in the TasAlyser software are possible; the list can be found e.g. in the parameter database in the sensor signal definition form.

`SetValueByName` has to be used after `Insert:` and before `Remove`. Note that all elements of the command have to be separated by spaces, comma or semicolon.

Setting values from file

Instead of setting values one by one using `SetValueByName:`, test stand control can generate a text file with multiple values and have that file imported into the TasAlyser results. The command is

```
SetValuesFromFile: Filename
```

The ‘Filename’ can be an absolute path, or relative to a default directory which is defined in the Plugin settings in TasAlyser.

The file given in the command is a text file which contains one value per row with the same syntax as for `SetValueByName` (see above):

```
ValueKey TestStep value (unit) (limit) (position)
```

ValueKey (name) and *Test Step* are defined in the Discom parameter database. *Unit*, *limit value* and *position* value are optional or can be replaced by a dash – if not used. The elements of a text row are separated by whitespace, comma or semicolon. Here are examples for valid rows (assuming the existence of the used names for Value Keys and Test Steps):

```
Fir1     3-D   17.5  -  20.0
Fir1, 3-C, 22,7; -; 25
OilTemp Function 89.4 °C
```

Empty lines are allowed, and lines starting with a semicolon ; are considered to be comment lines.

The command is acknowledged with 1 after the file has been successfully processed or with 0 if an error occurred. After successful processing, the file is deleted.

The text file can use either ASCII or UTF-8 (with BOM) text encoding.

The command can be used multiple times in each test run. If values occur twice, the last instance will overwrite the previous. Be aware that with the start of measurement for a test step, all previous values for that test step are deleted, including those inserted by SetValue commands. Therefore, use `SetValueByName` and `SetValuesFromFile` only after the according test steps have been measured, or for test steps which do no regular measurement.

Setting vectors from file

It is possible to set vector data (like a spectrum) using the file interface. (This feature is not available for the direct `SetValueByName` command.) To set a vector, start a line in the file with `$V` and use this format:

```
$V ValueKey TestStep x0 delta dlen x-unit y-unit
```

After this there have to follow exactly `dlen` lines with one floating point value per line.

x0 is the first x position of the vector (will be 0 or 1 in many cases), *delta* is the *x* spacing of the values, and *dlen* the total number of values. For example, a spectrum might have *x0*=0, *delta*=0.5, *dlen*=512, so the highest x value will be 255.5.

When specifying the *ValueKey* in the parameter database, you have to use for the “Instrument” clavis element an instrument which generates vector data, like Spectrum or Time Signal.

Extras

The GetValues plug-in knows two extra commands not directly related to measured values:

`EndMode:`	–	1: test step finished; 0: error
`LookupResource:`	`Source  Resource-Name`	Value line from resource file

The command `EndMode` finishes a test step without entering any other test step and is equivalent to the command `Mode: $Nil`

LookupResource queries a line from the application internal parameter files. Possible sources are “Application”, “Messages” and “Status”.

Trigger Parameters Plugin

In the measurement system, measurement of ramps over control values (like, for example, a speed ramp from 1500 to 3200 rpm) is controlled by so-called *triggers*. The trigger parameters (like the setting "speed ramp in 3rd gear drive goes from 1500 to 3200 rpm") are managed within the parameter data base.

Using the SetTriggerParams plugin, test stand control can change the trigger parameters. The command is `SetTriggerParams`.

As a second function of this plugin, the calibration file can be switched.

Command	Arguments	Reply

`SetTriggerParams:`	`trigger name`
	`test step`
	`start vaue`
	`end value`
	`step width`
	`sensor`
	`dimension`
		?: syntax error (e.g. missing argument)
		0: not executed (e.g. wrong name)
		1: parameters set
		(If the option 'Dpm42-Syntax' is switched on, the replies are preceded with `<R>SetTriggerParams:` .)

`SelectCalibration`	`Calibration file name (without file extension)`	?: missing argument 0: file does not exist 1: success

The first four arguments are mandatory; the last three can be omitted (from the back). Example:

```
SetTriggerParams: Standard 3-D 1500 3200 25
```

sets the trigger parameters for trigger "Standard" in test step "3-D" to a ramp from 1500 to 3200 with a step width of 25. The sensor is unchanged as set in the parameter data base and the "dimension" is implicitly 1. If the sensor must be given, the example could look like this:

```
SetTriggerParams: OrderTrack 3-D 1500 3200 25 InSpeed
```

Some important hints:

- It is only possible to *change* trigger parameters, but not to *create* them. Trigger parameters for any trigger name and test step have to be set up in the parameter data base before they can be changed with the SetTriggerParams command.
- The `SetTriggerParams` command must be sent at any time after the `Insert` command, but before the `Mode` command of the test step for which the parameters shall be set. (That is, you have to send "`SetTriggerParams: Standard 3-D` …" before you send "`Mode: 3-D`".)
- The parameters are valid until the end of the current test run (or until they are changed again with a different `SetTriggerParams` command). So if a test step should be repeated, it is not necessary to send the `SetTriggerParams` again.
- For rising ramps (speed drive ramps) the "start value" is less than the "end value" (like 1500 and 3200). For falling ramps (speed coast ramps), the start value is higher than the end value (as in `SetTriggerParams: Standard 3-C 3200 1500`).
- Names of triggers and sensors (if used) must match the names from the parameter database. Check the Trigger Parameters form there.

Caveat: the trigger parameters have an immediate influence on the measurements. So changing the trigger parameters will almost certainly lead to different measurement results (different values and curves) and thus produce different evaluation results!

SelectCalibration

The argument of the **`SelectCalibration`** command is the file name of the according calibration file, without file extension. The calibration files are typically located in the *Locals* folder within the project folder and are xml files. Example: the command

```
SelectCalibration: Calib-X1
```

activates the calibration stored in Calib-X1.xml.

The command must be used before the `Insert` command.

Sensor Configuration Plugin

In some projects, the sensors in use change between test runs. For example, there may be more than one noise sensor, but for each individual test, only one of these is used. In these situations, the project has a number of so-called *sensor configurations* defined in the parameter data base. It is possible to assign a sensor configuration with each test object type in the data base, so type-dependant sensor switching can be achieved solely by defining it within the parameter data base.

If the sensor configuration is not type dependant but is switched by the test stand, the test stand can tell the measurement system which sensor configuration to use for the next test run. There are three equivalent forms of this command:

Command	Arguments	Reply

`SetSensorConfiguration:`
`SetSensorConfig:`
`SensorConfiguration:`
	`configuration name`	0: not executed (e.g. wrong name)
		1: configuration set

The single argument of this command is the name of one of the sensor configurations defined in the parameter data base.

The command has to be sent *before the `Insert:`-Command* and is valid only for the following test run.

When this command is used, TasAlyser will automatically create an ‘additional info’ entry in the measurement result files containing the sensor configuration name which was selected.

The plugin supports one additional command which can be used in conjunction with the Signal Guard module. That module raises an alarm when a sensor signal peak value surpasses a given limit. This alarm can be used for an emergency stop of the test stand.

Command	Arguments	Reply
`SignalGuardSetActive:`	`0 / 1` `Off / On`	1

With command argument 0 (Off), Signal Guard is disabled until the command is used again with argument 1 (On) or until the next test run starts.

RecorderControl Plugin

Using this plugin and the `WaveRecorderControl` command, the recording of sensor signals in wave files can be controlled.

In normal operation, wave files are recorded automatically by the TasAlyser measurement application. For special applications where the automatic control is not applicable, this command can be used to control recording.

Command	Arguments	Reply
`WaveRecorderControl:`	`(Action code)`	1 or 0, meaning depends on command

These Action Codes (each one in three variations, as number or name) are available:

Aktion code	Action
`1 / AUTO / Auto`	Switch on or off automatic recording by TasAlyser. A second argument specifies which: 1/ON/On or 0/OFF/Off. Reply is the previous state. If no second argument is given, the reply value shows the current state without changing anything.
`2 / REC / Rec`	Status request: is a recording under way (reply 1) or not (reply 0)?
`3 / START / Start`	Starts a new recording. This is only possible within a test run (after "Insert:"). The recording ends automatically with the test run ("EndOfTest:") and can be ended at any time with action code 6. As second, optional argument a text can be given which will be inserted into the file name.
`4 / PAUSE / Pause`	Pause recording.
`5 / CONT / Cont`	Continue recording.
`6 / END / End`	End recording and save wave file.
`7 / DEL / Del`	Stop recording and delete wave file.

Actions 4 to 7 interact with TasAlysers automatic wave recording. For example, action code PAUSE will also pause an automatically started recording, not only those started with action code START. Action code END stops any recording, independent on how it was started. The reply to actions 3 to 7 is 1 if the action was performed successfully, and 0 if not (e.g., PAUSE when there was no recording running).

Action code START has no effect when there is already a recording under way.

Usage example: a recording is started, with text "MySong" getting part of the file name. Then recording is paused, later continued and finally ended.

```
WaveRecorderControl: REC MySong
WaveRecorderControl: PAUSE
WaveRecorderControl: 5
WaveRecorderControl: End
```

Within a test run, any number of recordings can be done. The file name of a recording is built according to the rules specified in the WaveRecorder module in TasAlyser. The time stamp element of the file name will be the time when recording started, not the time when the test run started (as for automatic recordings).

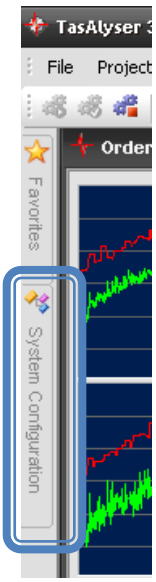
Appendix: Serial, Profibus and UDP Communication

As was described in the first part of this documentation, the Rotas system and test stand control (called ‚PLC' in the following) always communicate by the exchange of text messages.

This exchange can be done using various physical connections. The most simple, robust and most widely used method is a standard RS232 serial connection. Another widespread method is Profibus communication. This appendix contains hints regarding the commonly used communication methods.

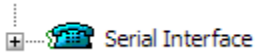
Serial Communication

In spite of (or because) the serial connection being from the old days of computer technology, it is hard to beat regarding simplicity and robustness. This success story found its continuation in the Universal Serial Bus – USB – which is also used for connecting the TAS hardware to the measurement PC. Even modern computers without built-in serial port can easily be equipped with a serial-to-USB converter. And transfer speed is no issue with current day computers: the short text messages and commands for measurement control are transferred virtually instantaneous.

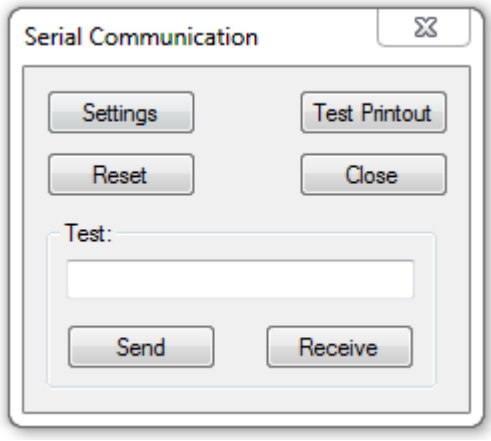


Settings

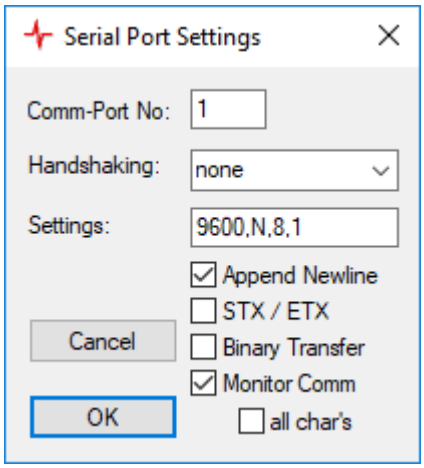
Setup of serial communication in the measurement program is quite simple: open the docking window **System Configuration**, expand the **Evaluation** branch and within it the node **Command Center** (see also the picture on page 14). Among the sub-nodes of that branch you will find the module for serial communication (with the picture of an old-style telephone):



Double-click on the entry to open the window of this module:



Press the **Settings** button to set up the parameters for the serial port (port number, baud rate and more). Standard settings are 9600 Baud, no parity, 8 data bits, 1 stop bit, no handshaking:



Reset does a reset of the selected serial port. You will get an error message at TasAlyser startup, if the selected serial port is not available.

To test the communication, enter a text into the box below **Test:** and press **Send**. The text will be sent out to the serial port. Pressing **Receive** will simulate an incoming message.

In the docking window "Output" you can monitor the serial communication and read all tests being sent out or coming in (see page 3).

Basic serial interface test

If the serial communication does not work out of the box, you can use the extra software tool **VcTerm** to do more tests. It is located in the Discom installation folder, which usually is C:\Program Files (x86)\Discom\bin.

Quit the TasAlyser program to free the serial port and then start VcTerm.exe. In the menu **CommPort** you can set the serial port properties and then open the port. In the main window of VcTerm, all texts coming in are echoed, and each line of text you type and terminate by pressing `<Enter>` is sent out via the serial port.

Profibus/Profinet Communication

For Profibus/Profinet communication, the measurement PCs are equipped with a CIF Profibus or CIFx Profinet card from the company Hilscher. We will send you the corresponding GSD file upon request.

First, you have to set up the card. This is done using the program **Sycon** (Profibus) or **netx** (Profinet) from Hilscher. Set up the card using the following information:

- Adresses and areas for communication have to be set in the way determined by the PLC. "Input" and "Output" are from the PLC point of view. If in the PLC "Input" is defined first, it has to be the same in the Sycon program.
- The communication buffer must be set to sufficient size. If you do not plan to transfer complete measurement report texts, a buffer size of 64 bytes is well enough. The buffer data type (bytes, words, double words) does not matter[3] but has to be set the same on both sides.
- Communcation master: you have to enter the address of the Profibus master (usually the PLC). It is not important, which card type you select here.

For further information please refer to the Hilscher documentation.

Communication between PLC and TasAlyser works by exchanging text messages written into the Profibus buffers. The buffer is filled using the following layout:

Byte 0	Byte 1	Byte 2	Byte 3	…					
counter	text length including 0 byte	message text (no cr/lf at the end!)					0 (NULL)		

The sending communication partner first writes the message text into the buffer, starting at byte 2. The text has to be terminated by a zero byte. In buffer byte 1 the total length of the message is stored, including the zero byte.

Please note that the text *may not contain zero bytes*. Example: if the text buffer contains the byte values 65, 66, 67, **0**, 68, 69, 0 and the length byte the value 7, TasAlyser will still only see the text "ABC", because the zero byte terminates the text. The text buffer should be initialized with byte values 32 (space) if necessary. Buffer content 65, 66, 67, **32**, 68, 69, 0 will be understood as "ABC DE".

Now there should be a short wait time (25 to 50 milliseconds, one PLC cycle or similar), during which the Profibus will transfer the complete buffer contents.

Finally, the counter in buffer byte 0 is increased. (After reaching 255 the counter will restart at 0; only *changing* the counter matters.) The counter change validates the completeness of the message and signals the communication partner to start processing it.

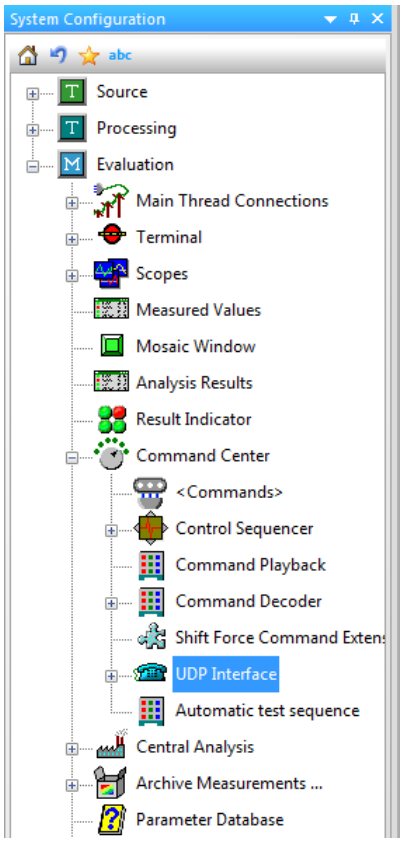
Note: in the settings of the Profibus communication module in the TasAlyser software the use of the counter can be switched off. If this was accidentally done, there will appear one or two additional characters in front of the message text.

[3] Please note: the *contents* of the buffer is always the same, independent of the selected data type.

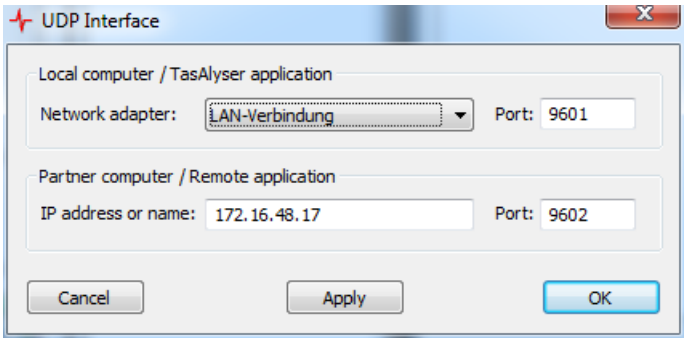
UDP Communication

The test stand software sends UDP packets targeted to the measurement application, and receives UDP packets as answers. The data contained in the packets are the command texts (8 bit ASCII), followed by a terminating Null character. The data length of the packets is thus equal or greater than the command text length + 1. The measurement computer's replies ar of the same format.

In the TasAlyser measurement application, the IP address of the partner computer as well as the port numbers have to be set up. To do this, open the **System Configuration** window, expand the **Evaluation** node and the sub-node **Command Center**:



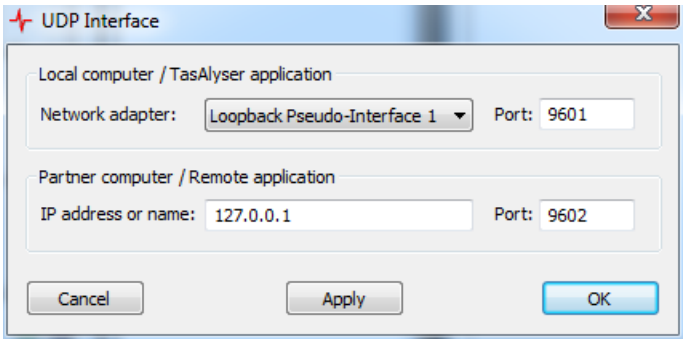
Double-click on **UDP Interface** to open the settings window for this module:



In the upper part **Local computer / TasAlyser application**, you select the network adapter (see next section) and local port number (see comment below).

In the lower section **Partner computer / Remote application**, you have to enter the IP address of the test stand control computer and the port number on which it listens for TasAlyser's replies. You can use the test stand control computers name instead of the IP address (but without any \\ or similar).

If TasAlyser and the test stand control software are running on the same computer, the settings may look like this:



Here, again, you may use the name `localhost` instead of the IP address `127.0.0.1`.

The local communication port number 9601 can be changed if necessary. It has been chosen with care and usually should be free – please look up "List of TCP and UDP port numbers" in Wikipedia: http://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers.

TasAlyser will start listening for commands on this port right after the application has been started and is ready.

Finding the right network adapter

The measurement computer can be equipped with several network adapters: for example, two LAN cards, WiFi, a VPN adapter and others. One of these has to be selected for UDP communication.

The UDP Module's settings window lists all available adapters (limited to those configured for IPv4, because TasAlyser uses IPv4 only).

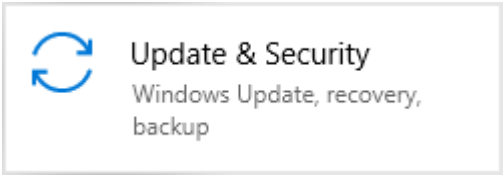
To get more information about the available network adapters, open the measurement computer's **Control Panel** and go to the **Network and Internet** section. Open the **Network and Sharing Center**. On the left hand side, in the list of available commands, you will find **Change adapter settings**. Click on this command, and you will be presented with the list of existing network adapters. Look up the settings for each adapter to find the one which connects to the test stand computer.

After choosing a new network adapter in TasAlyser, Windows Firewall may become suspicious. You may see the warning that it has blocked TasAlyser and the request to allow network communication for TasAlyser. This you have to grant to make UDP communication possible.

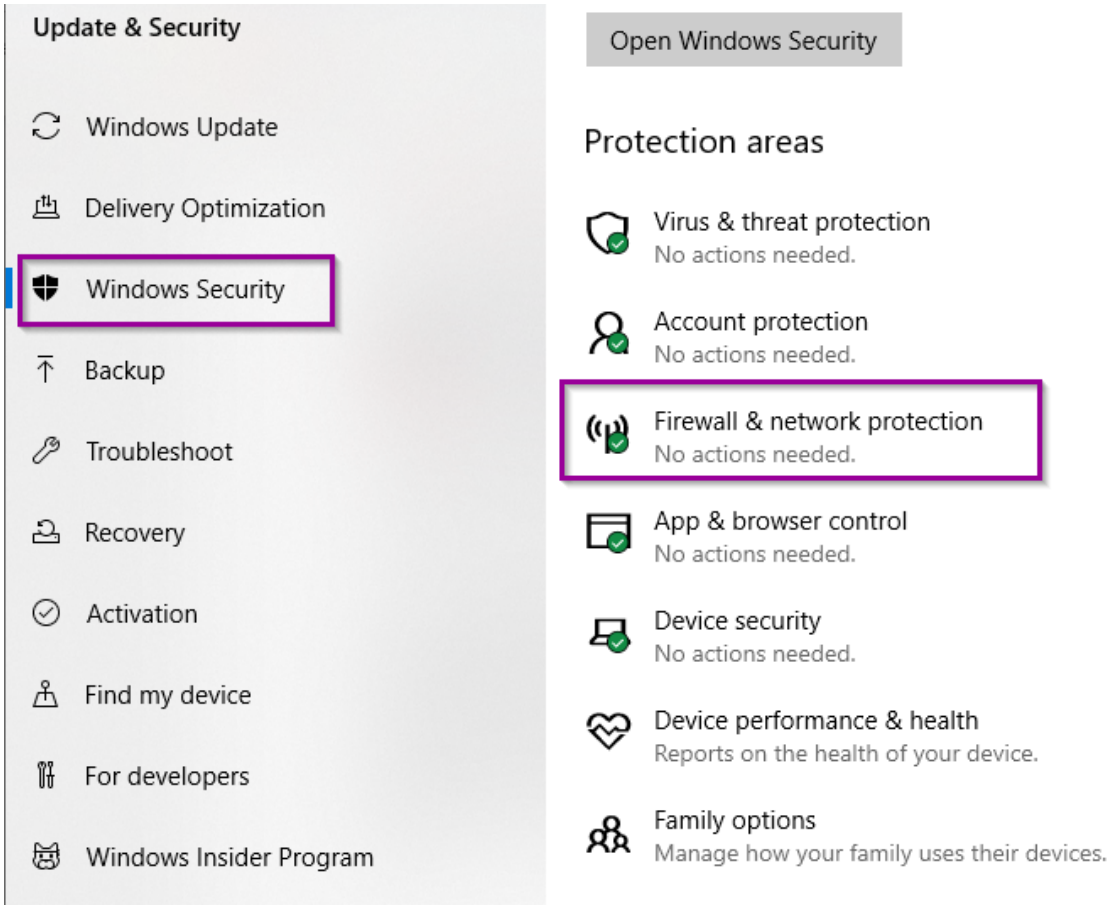
Checking the Firewall

The Windows Firewall will by default block any unknown application which tries to establish a network communication using non-standard ports (like port 80, which is used for http). When TasAlyser is started on a computer for the very first time with any project that contains UDP communication, the Windows Defender will pop up only once and ask whether this application shall have permission to communicate through the firewall.

When UDP communication does not work although all settings seem to be correct, check the Firewall settings. Open the Windows Control Panel and go to the Security section:



In that section, select "Windows Security", and then from the option on the right side "Firewall & Network Protection":



Inside “Firewall & Network Protection”, click on “Allow an app through firewall”. This will open a new window with a list of applications using network communication. Scroll down to TasAlyser and set the check mark at the according network type – in most cases, this will be the “private” network: