



Programmer's Manual

Revision 2022.10.10

Table of Contents

1	Introduction	5
2	Application configuration	5
2.1	Basics	5
2.1.1	General	5
3	Login system	6
3.1	Permission levels	6
3.2	Standard login	6
3.2.1	User account list	6
3.2.2	Custom rights	7
3.2.2.1	Definition	7
3.2.2.2	Usage	7
4	Command-line arguments	8
4.1	Install	8
4.2	Login at startup	9
4.3	Startup project	9
4.4	Startup test-file	9
5	Teststation	9
5.1	IO and MX mapping	9
5.1.1	Supported devices	11
5.1.2	Aliases and segments	12
5.1.3	IO commands	13
5.1.3.1	*rst (Reset)	13
5.1.3.2	seg (Set segment)	14
5.1.3.3	s (Set output)	14
5.1.3.4	c (Clear output)	14
5.1.3.5	ca (Clear all non-segmented)	15
5.1.3.6	r (Read input)	15
5.1.3.7	ra (Read all non-segmented)	15
5.1.3.8	d (Delay)	16
5.1.4	MX commands	16
5.1.4.1	*rst (Reset)	16
5.1.4.2	route/croute (Set routing)	16
5.1.4.3	seg (Set segment)	17
5.1.4.4	set/cset (Set TP)	17
5.1.4.5	clr (Clear TP)	18
5.1.4.6	d (Delay)	20
6	Testprogram	20

6.1	Asynchronous program flow	20
6.1.1	Commands	21
6.1.1.1	#callasync (Call a function asynchronously)	21
6.1.1.2	#sync (Async function synchronization)	21
6.1.1.3	#async (Async functions control)	22
6.1.1.4	#asyncflag (Async notification system)	22
6.2	Variables	24
6.2.1	Standard variables	24
6.2.2	Test-file variables	25
6.2.3	Arrays	25
6.2.3.1	Definition	25
6.2.3.2	Length	25
6.2.3.3	Usage	25
6.2.4	Commands	26
6.2.4.1	#cnt (Counter operations)	26
6.2.4.2	#var (Variable operations)	27
6.3	Interrupt events	28
6.3.1	Event types	29
6.3.1.1	prj-closing (Project closing)	29
6.3.1.2	prj-closed (Project closed)	29
6.3.1.3	oper-changed (Operator changed)	29
6.3.1.4	run-at (Run at specified time)	29
6.3.1.5	run-every (Run every interval)	30
6.4	Commands	31
6.4.1	Syntax	31
6.4.2	Summary	35
6.4.3	Flow control	35
6.4.3.1	#goto (Jump to label)	35
6.4.3.2	#call (Call a function)	36
6.4.3.3	#return (Return from function)	36
6.4.3.4	#throw (Exit function with error)	37
6.4.3.5	#onerror, #onfail (Check for error or fail)	37
6.4.4	Operator interface	38
6.4.4.1	#msg (Show a message)	38
6.4.4.2	#dlg (Show a dialog)	40
6.4.4.3	#status (Set test status of current panel)	43
6.4.4.4	#resultlist (Result list operations)	44
6.4.4.5	#panel (Panels control)	45
6.4.4.6	#stopwatch (Integrated stopwatch control)	46
6.4.4.7	#userbtn (User-button control)	48
6.4.4.8	#adjust (Value adjust dialog)	50
6.4.5	IO	52
6.4.5.1	#catchio (Wait for a specified IO device input)	52
6.4.5.2	#oninput (External input interrupt)	53
6.4.6	Files	54
6.4.6.1	#file (File operations)	54
6.4.6.2	#export (Export to file)	59

	6.4.6.3	#stat (Statistics)	60
	6.4.6.4	#statrec (Basic statistics record)	62
	6.4.6.5	#statsave (Basic statistic save)	62
6.4.7		Test-file	63
	6.4.7.1	#cellread, #cellwrite, #cellerase, #cellcopy (Cell direct-access)	63
	6.4.7.2	#retclear (Clear return values)	65
	6.4.7.3	#testfile (Testfile control)	66
	6.4.7.4	#str (String operations)	67
	6.4.7.5	Localization	69
6.4.8		Printing	71
	6.4.8.1	Labels	71
	6.4.8.2	Text	73
	6.4.8.3	Spreadsheet	74
6.4.9		Special	75
	6.4.9.1	#catch (Wait for a specified response of device)	75
	6.4.9.2	#login (Login operations)	75
	6.4.9.3	#bct (Batch-test)	77
	6.4.9.4	#extprocess (Run an external process)	79
	6.4.9.5	#dummy (Dummy-test control)	81
	6.4.9.6	#get (Get a value)	83
6.5		Statistics	84
	6.5.1	Automated statistics	84
	6.5.1.1	AMSTAT sheet	85

1 Introduction

FunTEST is the functional testing software from FPC LLC.

It allows to create testing sequences for the target application. The core of the sequencer is based on the OpenOffice Calc sheet. This specifically formatted sheet is used to store test-program steps and runtime data. The funTEST reads the sheet, processes the data and writes return values back.



This manual describes some funTEST functional blocks and especially test-file commands usage.

2 Application configuration

To configure funTEST, log as an administrator and click "Administrator" button on the main window.

Test-station

All testing and measuring devices are unified into the test-station. FunTEST does not access hardware directly, this is realized by **device plug-ins**. To access any instrument the corresponding plug-in is needed. Each plug-in runs within its own process (container process) to ensure maximum stability of the testing system.

Project

Project contains a set of test-files (programs). A test-station must be assigned to the project - this will select devices to be used by loaded test-file.

2.1 Basics

Switch to Basics tab, on this tab funTEST's general behaviour can be changed.

2.1.1 General

Language

funTEST language can be selected here. Application must be restarted to take effect.

Login type

Login defines rights of logged user. Login system supports plug-ins, login procedure can be customized to match customer's requirements (e.g. login using card-reader, database login, etc...). By default, the standard login is selected. This default plug-in uses local encrypted file to store user accounts. **funTEST cannot start without a valid login plug-in**. If any other selected plug-in failed to load, funTEST will try to load a standard plug-in.

Optimize for touchscreen

Some controls, especially in the operator interface can be optimized for a touch-screen. When checked, these controls are enlarged.

3 Login system

funTEST has implemented a file-based logging system to monitor running events. Log files are stored inside funTEST logs subdirectory like simple text file with .log extension (or compressed .gz for older). From application, log window can be shown by click on "System" menu in the right-bottom corner → "Show logs" item.

Inside the "Logging system" section the logging system can be globally disabled, enable to log also debug entries, set limit of count of log entries in the computer's memory and set parameters of logging to file.

3.1 Permission levels

In the funTEST there are 3 basic permission levels:

Permission name	Permission key	Description
Administrator	adm	Access to global configuration, project definition, test-file definition and can add test-files.
Programmer	prg	The programmer cannot change the funTEST's configuration, but can open the test-file and edit the test-file.
Operator	oper	Operator is not allowed to edit anything, operator can only select the project and testfile and run in - the production process.

Administrator and Programmer is not allowed to start the test. Each permission is separated level, not the higher level and it does not include rights of another level. This means the Administrator level does not include the Programmer and Operator level and Programmer does not include the Operator level. User with full access = Administrator + Programmer + Operator.

Each permission level is identified by a key - `adm`, `prg` and `oper`. This key is a reference for the test-file and it's used for queries.

3.2 Standard login

In this chapter, the default **standard login plug-in** (local file-based) is described.

Use only password to login

When checked, the user name will not be required during login. Make sure, that every user has a unique password set.

3.2.1 User account list

The list shows all user accounts. There you can add a new user, modify or delete the existing one.

New user account

To add a new user, click the "New user..." button below the list. The "New user" dialog will pop up. Type a user name, login name (this must be unique), password (cannot be empty) and select account rights (at least one must be checked). Click "OK" to confirm new user account.

Edit existing account

In the list select desired account and click the "Edit" button below the list. The "Edit user" dialog will pop up. Any value can be changed. If you do not want to change the password, leave it blank.

Note: the administrator account is only possible to change the password

Delete existing account

Select desired account, click "Remove" and confirm.

Note: the administrator account cannot be removed

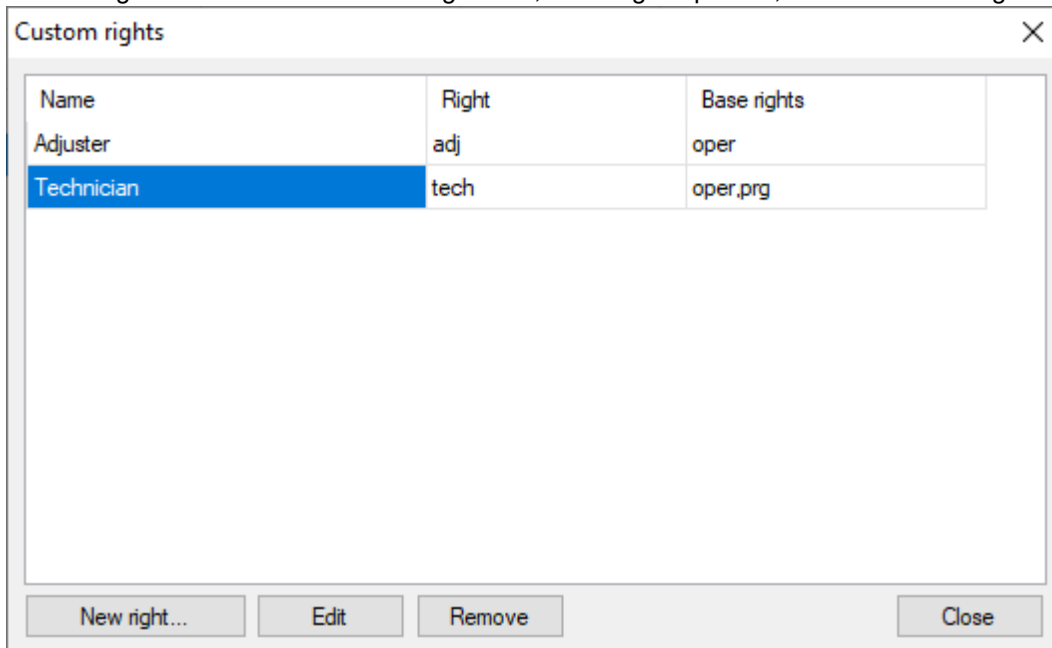
3.2.2 Custom rights

The standard login plug-in support custom rights.

The custom right allows to define additional rights. This allows permission extension of identification in the test-file. They are always based on [basic permission levels](#) (base rights). Base rights define the behaviour of additional right in the funTEST (because the funTEST uses 3-level permission system). The resulted permission will be base-rights + additional right.

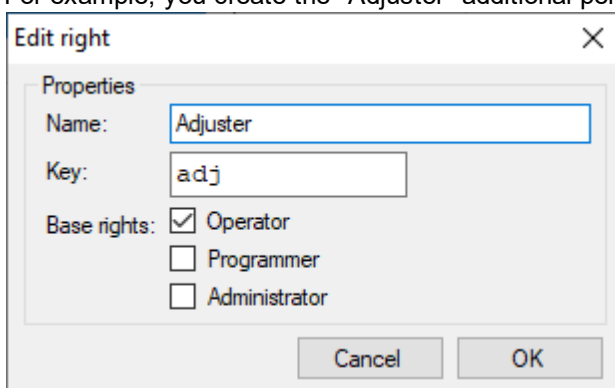
3.2.2.1 Definition

Custom rights can be defined via configuration, tab "Login Options", button "Custom rights...":



Dialog with custom rights list

When adding new custom right, it's always necessary to define "base rights" of the additional one. For example, you create the "Adjuster" additional permission, with "Operator" like base rights:



Additional right definition

The "Adjuster" will have the behaviour of "Operator" in the funTEST, but in the test-file you can identify the additional right. The key "adj" is the identification to the test-file - it's important property, make it unique and as short as possible (typically shorten word, or two shorts with a hyphen "-" and so on).

3.2.2.2 Usage

Definition

All created custom rights will be shown in the user definition dialog:

Selection of addition right in the user definition

It's possible to select only the additional right, because the base rights are defined in the additional permission. Of course in this dialog you can combine it if it's necessary.

Test-file identification

It's possible to identify the additional rights in the test-file in the IDENTIFICATION section of HEAD:

6	IDENTIFICATION	
7	Project name	Dev
8	Station name	Dev
9	Operator name	Worker1
10	Login name	wrk
11	User rights	adj,oper
12	TestFile name	test

In this place will always be the combination of additional + base rights.

4 Command-line arguments

It is possible to run funTEST with command-line arguments to automatically i.e. login or start project immediately after startup.

Usage:

```
funTEST.exe [arguments]
```

4.1 Install

```
--install
```

Perform initialization:

- Create application's data directory structure
- Create default user-definition file of standard login plugin

Warning: previous user-definition file of standard login plugin will be overwritten!

4.2 Login at startup

```
--startup-login <username>: <password>
```

Login to funTEST via command-line argument. Do not show the login dialog while funTEST is starting.

Parameters

username	[string]	Username to be used
password	[string]	Password of specified user

Examples

```
--startup-login "fpc:secret"  
Auto-login using "fpc" user with "secret" password.
```

4.3 Startup project

```
--startup-project <project-name>
```

Auto-select the project when funTEST starts. When no start-up testfile is selected, the test-file selection dialog of specified project will appear.

Parameters

project-name	[string]	Project name to be selected
--------------	----------	-----------------------------

Examples

```
--startup-project "demo"  
Use "demo" like a funTEST startup project.
```

4.4 Startup test-file

```
--startup-testfile <testfile-name>
```

Auto-select the test-file when funTEST starts. Startup project is required to select the test-file.

Parameters

tesfile-name	[string]	Test-file name to be started
--------------	----------	------------------------------

Examples

```
--startup-project "demo" --startup-testfile "test"  
Set "test" test-file from project "demo" like a startup project.
```

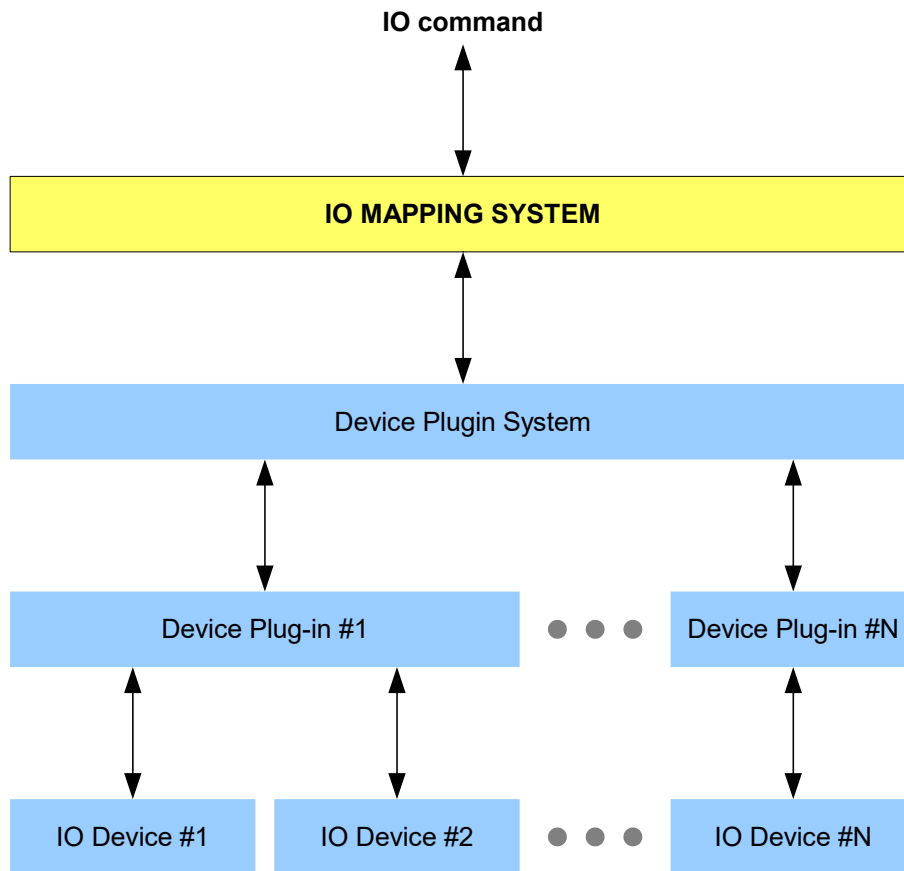
5 Teststation

5.1 IO and MX mapping

IO mapping

The IO mapping in then funTEST's test-station provides an **unification access** to IO devices. In the application can be more than one IO interface, with its own configuration and access. The IO mapping system finds all supported IO devices and **joins inputs/outputs** to only **one virtual IO device**. It does not touch the access to the hardware, every IO plugin remains to be controlled by its own plugin.

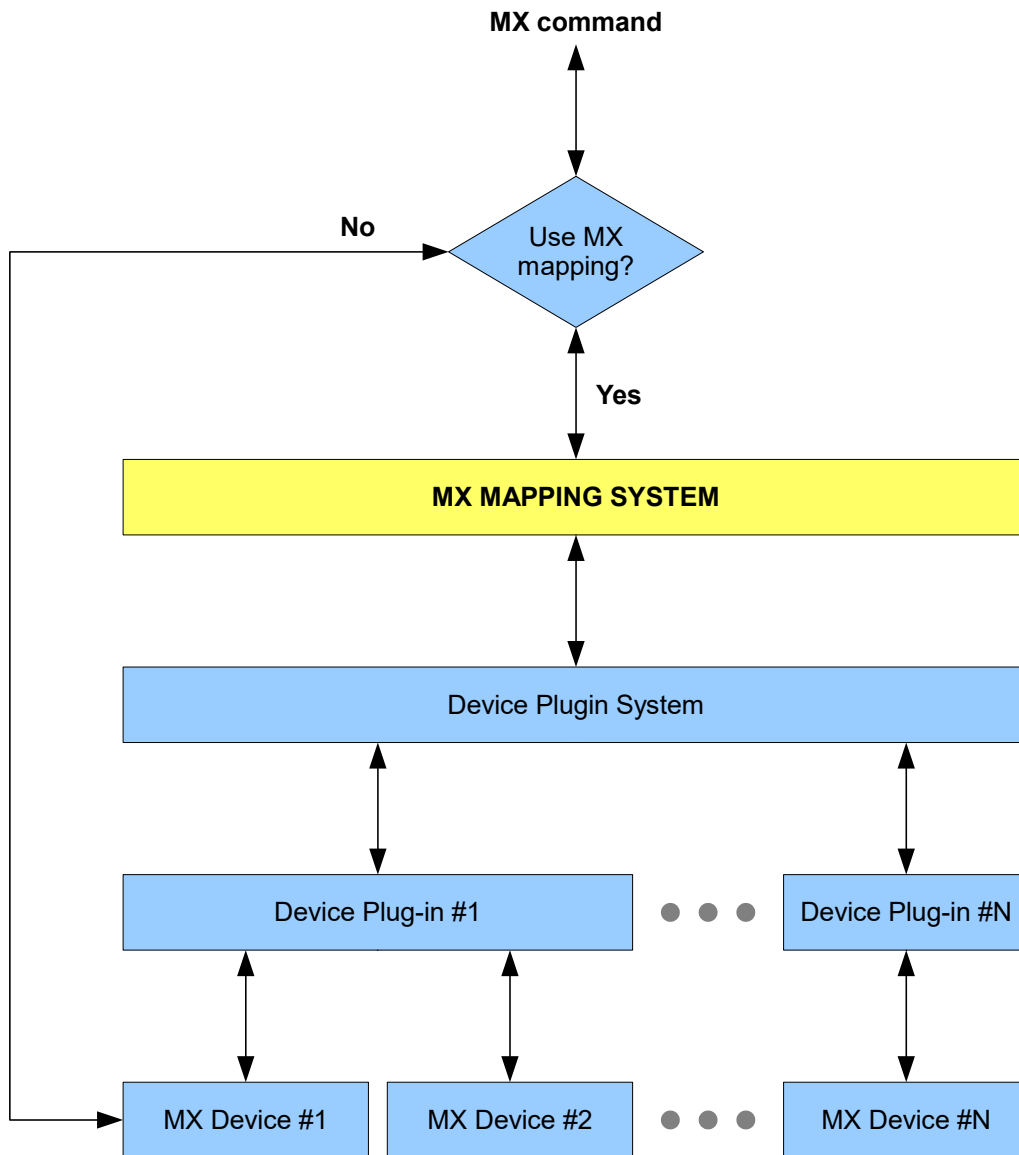
The IO mapping is required for use the dedicated IO column in the test-file.



MX mapping

The MX (matrix) mapping is similar to IO mapping. MX mapping system finds all supported MX devices and joins test-points together to only one virtual MX device.

The MX mapping is required for use the dedicated MX column in the test-file.



Against IO mapping, there is a possibility to bypass the MX mapping. In this case, only first MX device in the test-station will be used to control by MX column. Commands are directly forwarded to this device. Aliases and segmentation are not used.

5.1.1 Supported devices

IO devices

Vendor	Model	Connection	Inputs	Outputs
FPC	USB 8I8O	USB	8	8
FPC	USB 16I16O	USB	16	8
FPC	Relay Boards CFPC-108 and CFPC-138	RS-232/USB	-	40
FPC	MatrixBox MX2400 IO	Ethernet	16	16
Advantech	PCI-1730	PCI	16	16
Advantech	PCIE-1730	PCI-Express	16	16
Papouch	Quido RS 8/8	RS-232	8	8

Matrix devices

Vendor	Model	Connection	Max TP	Routing support
FPC	MatrixBox MX75	USB	75	-
FPC	MatrixBox MX100, MX400/4	USB	100	Yes
FPC	MatrixBox MX400	USB	400	Yes
FPC	MatrixBox MX2400 MX	Ethernet	544	Yes

Devices listed above are recognized by funTEST as IO/MX device and included to mapping system. Support of connected device by the mapping system is indicated by **IO** or **Matrix** in the **Type** column in the definition of test-station, tab "Devices":

Enabled	Serial number	Alias	Name	Type	Connection
<input type="checkbox"/>	1430201412050002	1430201412050002	MatrixBox MX2400	Generic	Plug-in system
<input checked="" type="checkbox"/>	1430201412050002IO	mx-io	MatrixBox MX2400:IO	IO	Plug-in system
<input checked="" type="checkbox"/>	1430201412050002MX	mx	MatrixBox MX2400:MX	Matrix	Plug-in system
<input type="checkbox"/>	1430201412050002MM	1430201412050002MM	MatrixBox MX2400:MM	Generic	Plug-in system
<input checked="" type="checkbox"/>	16IO0026	usb-io	USB 16I16O	IO	Plug-in system

5.1.2 Aliases and segments

For every input/output (IO mapping) or test-point (MX mapping) is possible to define alias and segment. Using aliases you can name every input/output or TP and use them in the test-program instead of just numbers. This makes the test-program as clear as possible and it is highly recommended.

Using segments, it is possible to divide inputs or outputs to logical groups. Segments enables you to define more pins with the same alias and divide them to groups, for example for multipanel testing.

IO segment example

MX segmenting is used by the same way.

Output nr.	Alias	Segment
0	gnd	1
1	vcc	1
2	gnd	2
3	vcc	2

We have a dual-panel test, where two identical boards has to be tested. Every board has power supply, which has to be connected before functional testing. This means, the command to **SET** output must be called on IO mapping using "gnd" and "vcc" aliases. But there are two "gnd" and two "vcc" defined:

Without segments

These aliases would have to be renamed to unique names, for example "gnd1" and "vcc1" for board A and "gnd2" and "vcc2" for board B. In the test-program also must be two blocks with IO command, separate for board A and B.

Using segments

There can be more pins with same alias defined. In the test-program you will use the same IO command both

for board A and B. The difference is in specifying segment ([seg](#) command), before the IO command will be executed. When you for example set segment to "2" and call command to set "gnd" and "vcc", the outputs 2 and 3 will be set.

Segments makes the program more easier, especially when multi-panel testing with a lot of same boards.

5.1.3 IO commands

Commands to control IO mapping system.

IO mapping system accepts more commands in a chain at once.

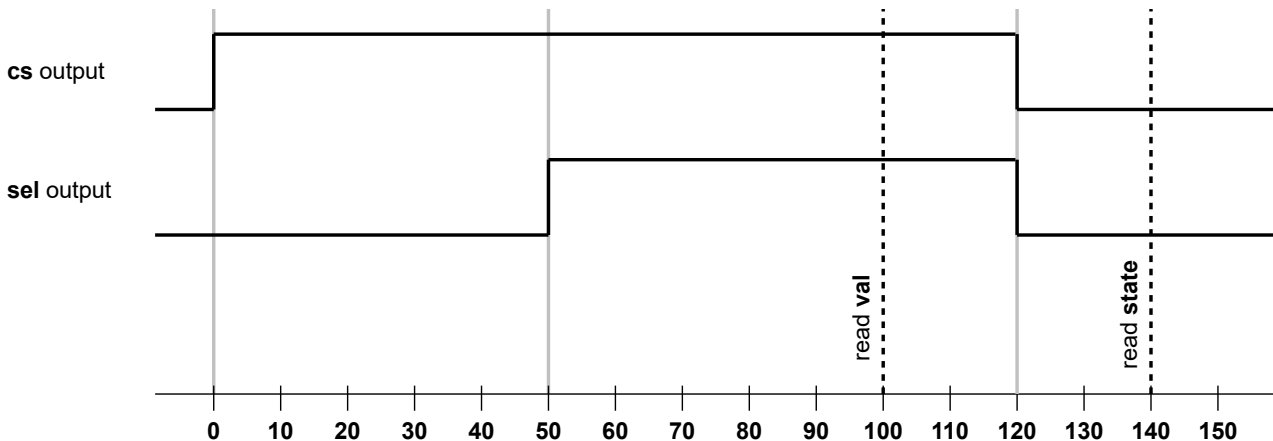
Example

```
s: cs: d: 50: s: sel: d: 50: r: val: d: 20: c: cs: sel: d: 20: r: state
```

This is a regular command to IO mapping system, that will do following:

- [Set](#) "cs" output(s)
- [Delay](#) for 50 ms
- Set "sel" output(s)
- Delay for 50 ms
- [Read](#) "val" input(s)
- Delay for 20 ms
- [Clear](#) "cs" and "sel" output(s)
- Delay for 20 ms
- Read "state" input(s)

Return value will contain states of "val" inputs and "state" inputs in order as they has been read.



5.1.3.1 *rst (Reset)

```
*rst
```

Clear all outputs (without applying outputs) and clear current segment setting.

Parameters

No parameters

Return value

No return value

5.1.3.2 seg (Set segment)

```
seg: <seg0>{: <seg1>: . . . : <segN>}
```

Set currently active segments. Previous settings is cleared.

Parameters

seg [int] Number of segment, range 1..N

Return value

No return value

Examples

```
seg: 1: 3
Set currently active segments to 1 and 3.
```

```
seg: *
```

Clear all currently active segments.

Parameters

No parameters

Return value

No return value

5.1.3.3 s (Set output)

```
s: <out0>{: <out1>: . . . : <outN>}
```

Set specified output(s) to ON state. The command reflect current segment settings.

Parameters

out [int] or [string] Pin number or pin alias of output

Return value

No return value

Examples

```
s: 3: 7: gnd: vcc
Set outputs with numbers 3 and 7 and all outputs with aliases "gnd" and "vcc" in currently active segment (s).
```

5.1.3.4 c (Clear output)

```
c: <out0>{: <out1>: . . . : <outN>}
```

Set specified output(s) to OFF state (clear). The command reflect current segment settings.

Parameters

`out` [int] or [string] Pin number or pin alias of output

Return value

No return value

Examples

```
c: 3: 7: gnd: vcc
```

Clears outputs with numbers 3 and 7 and all outputs with aliases "gnd" and "vcc" in currently active segment(s).

5.1.3.5 ca (Clear all non-segmented)

```
ca
```

Set all outputs to OFF state, without applying segment settings.

Parameters

No parameters

Return value

No return value

5.1.3.6 r (Read input)

```
r: <in0>{: <in1>: . . . : <inN>}
```

Read state of specified inputs(s). The command reflect current segment settings.

Parameters

`in` [int] or [string] Pin number or pin alias of input

Return value

1 = ON

0 = OFF

When more inputs are passed, results are separated by a color ":" (i.e. "0: 1: 1: 0: 1")

Examples

```
r: state
```

Read all inputs with alias "state" in currently active segment(s).

```
r: 4: 8: 12
```

Read inputs with numbers 4, 8 and 12.

5.1.3.7 ra (Read all non-segmented)

```
ra
```

Read all inputs, without applying segment settings.

Parameters

No parameters

Return value

Input states 0 or 1, separated by a colon ":".

For example for total of 8 inputs (in order 0 to 7): 0: 1: 0: 1: 1: 0: 0: 1

5.1.3.8 d (Delay)

```
d: <delay>
```

Delay for a number of milliseconds. Using this command it is possible to generate for example a pulse using only one command.

Parameters

delay [int] Number of milliseconds to delay.

Return value

No return value

Examples

```
s: led: d: 500: c: led
```

Set output(s) with alias "led" for 500 ms.

5.1.4 MX commands

Commands to control MX mapping system.

MX mapping system accepts more commands in a chain at once. The same like IO commands.

Example

```
route: imeas1: on: cset: 1: 10: 20: h: 30: 40
```

Will do:

- Set current measure #1 (MX400 series) ON
- Clear previously connected TPs, and connect TPs 10 and 20 to L-bus and TPs 30 and 40 to H-bus

5.1.4.1 *rst (Reset)

```
*rst
```

Disconnect all TPs and clears all currently active segments.

Parameters

No parameters

Return value

No return value

5.1.4.2 route/croute (Set routing)

This command requires a device with [routing support](#). Arguments depend on physically connected device - MatrixBox MX400-series has different routing than MX2400-series. For commands, see the route command reference for the specific device.

This is not related with MX mapping (test-points).

```
route -or- croute{:<device>}:<command>
```

The "croute" variant of command is the same like "route", but it clears previous routing settings before setting new one.

Parameters

device	[string]	Alias of target MX device to set routing. Required if there is more than one MX device in the test-station.
command	[string]	Hardware-dependent route command to specified device.

Return value

No return value

5.1.4.3 seg (Set segment)

```
seg:<seg0>{:<seg1>:...:<segN>}
```

Set currently active segments. Previous settings is cleared.

Parameters

seg	[int]	Number of segment, range 1..N
-----	-------	-------------------------------

Return value

No return value

Examples

```
seg: 1: 3
Set currently active segments to 1 and 3.
```

```
seg: *
```

Clear all currently active segments.

Parameters

No parameters

Return value

No return value

5.1.4.4 set/cset (Set TP)

Connect specified test-point(s) to the L/H BUS.

The "cset" command is same like "set", but all previously connected TPs are disconnected first. This command uses the MX mapping - all TP aliases are replaced with corresponding TP numbers first. The current segment settings is also reflected.

```
set:<tpL>:<tpH>
```

Connect TP pair (low + high) to BUS.

Parameters

<code>tpL</code>	[int] or [string]	Test-point low, number or alias
<code>tpH</code>	[int] or [string]	Test-point high, number or alias

Return value

No return value

Examples

```
set: 10: 20
Connect TP10 to LOW bus and TP20 to HIGH.
```

```
set: gnd: vcc
Connect all test-points with "gnd" alias to LOW bus and all test-points with "vcc" alias to HIGH bus.
```

```
set: l: <tpL0>{:...: <tpLN>} : h: <tpH0>{:...: <tpHN>}
```

Connect specified list of test-points to LOW bus and specified list to HIGH bus.

Parameters

<code>tpL_N</code>	[int] or [string]	Test-point(s) low, number or alias
<code>tpH_N</code>	[int] or [string]	Test-point(s) high, number or alias

Return value

No return value

Examples

```
set: l: 10: 20: h: 30: 40
Connect TPs 10 and 20 to LOW bus and TPs 30 and 40 to HIGH bus.
```

```
set: l: gnd: 10: h: vcc: 20
Connect all testpoints with alias "gnd" and TP10 to LOW bus, and all testpoints with "vcc" and TP20 to HIGH bus.
```

```
set: c
```

Disconnect all TPs from the BUS.

*Obsolete command, not recommended to use. Replacement: `clr: *`*

Parameters

No parameters

Return value

No return value

5.1.4.5 clr (Clear TP)

Disconnect specified test-point(s) from the L/H BUS.

This command uses the MX mapping - all TP aliases are replaced with corresponding TP numbers first. The

current segment settings is also reflected.

```
set: <tpL>: <tpH>
```

Disconnect TP pair (low + high) from the BUS.

Parameters

tpL	[int] or [string]	Test-point low, number or alias
tpH	[int] or [string]	Test-point high, number or alias

Return value

No return value

Examples

```
clr: 10: 20
Disconnect TP10 from LOW bus and TP20 from HIGH.
```

```
clr: gnd: vcc
Disconnect all test-points with "gnd" alias from LOW bus and all test-points with "vcc" alias from HIGH bus.
```

```
set: l: <tpL0>{:...: <tpLN>} : h: <tpH0>{:...: <tpHN>}
```

Disconnect specified list of test-points from LOW bus and specified list from HIGH bus.

Parameters

tpL _N	[int] or [string]	Test-point(s) low, number or alias
tpH _N	[int] or [string]	Test-point(s) high, number or alias

Return value

No return value

Examples

```
clr: l: 10: 20: h: 30: 40
Disconnect TPs 10 and 20 from LOW bus and TPs 30 and 40 from HIGH bus.
```

```
clr: l: gnd: 10: h: vcc: 20
Disconnect all testpoints with alias "gnd" and TP10 from LOW bus, and all testpoints with "vcc" and TP20 from HIGH bus.
```

```
clr: *
```

Disconnect all TPs from the BUS.

Parameters

No parameters

Return value

No return value

5.1.4.6 d (Delay)

d: <delay>

Delay for a number of milliseconds. Using this command it is possible to put delay between connection/disconnection TP(s) in the one command.

Parameters

delay [int] Number of milliseconds to delay.

Return value

No return value

Examples

cset: 10: 20: d: 100: set: 30: 40

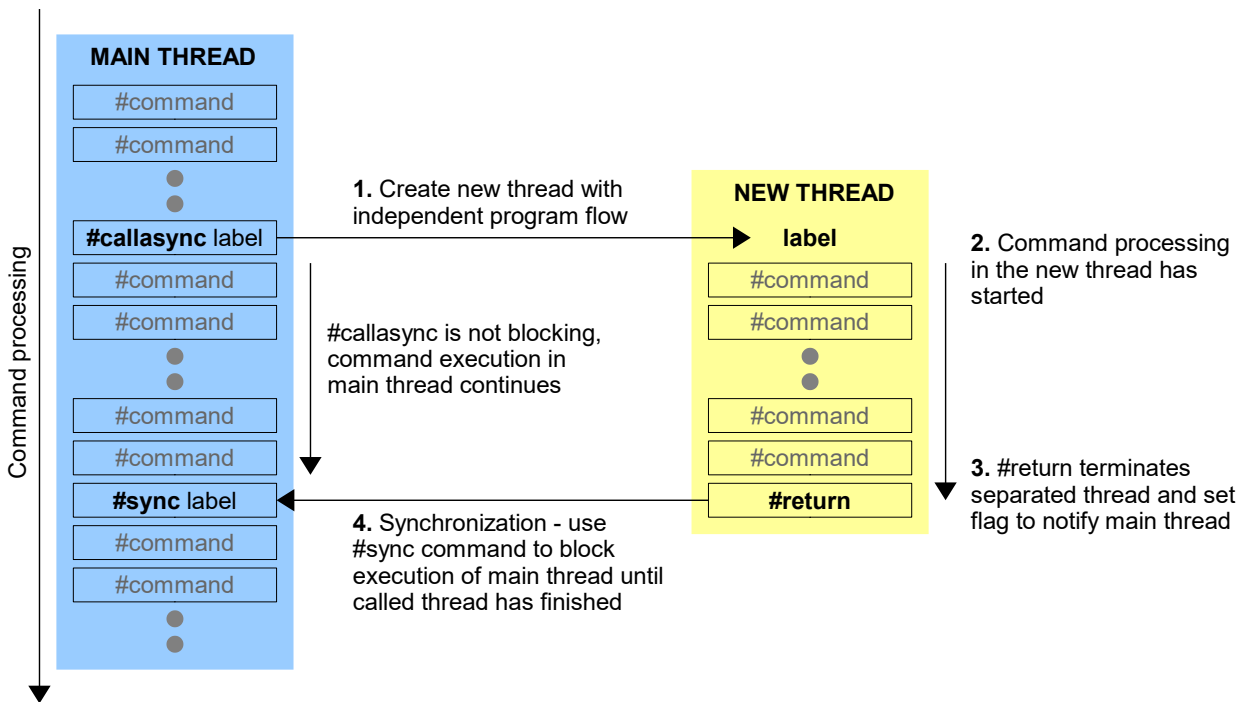
Connect TP10 to LOW, TP20 to HIGH, delay for 100 milliseconds and then connect also TP30 to LOW and TP40 to HIGH.

6 Testprogram

6.1 Asynchronous program flow

FunTEST since version 0.9.20709.0721 supports an asynchronous program flow in separated thread(s) - parallelism.

Functionality diagram



The benefit is possibility of **two or more independent command execution flows**. This allows to perform several sub-task at once. For example by using of two or more measurement instruments is possible to measure more parts simultaneously.

Control

To control asynchronous program flow the following commands are dedicated:

- [#callasync](#) - start a command processing of target function in the separated independent thread
- [#sync](#) - synchronize with asynchronously called function (wait until its ends), using this command it is possible to get the return value from async function
- [#async](#) - pause/resume/stop or get state of thread(s)
- [#asyncflag](#) - global flags (notification system) to for example block some device

6.1.1 Commands

6.1.1.1 #callasync (Call a function asynchronously)

```
#callasync | <function-name>{:<param0>:...:<paramN>}
```

Call a function asynchronously in the new separated thread. Target function is represented by a label. Function is a block of code begins by the label and ends by the [#return](#) (or [#throw](#)) command. No return value is stored to the stack, **the command is not blocking and it will not return to caller-line**. The programmer has to use the [#sync](#) command from caller thread to wait until called function is done (and optionally obtain return value or error).

Starting funTEST version 1.0.1906.311 multiple parameters are supported. Previous versions support param₀ only.

Parameters

function-name	[string]	Function name to run in the new thread
param	[string]	Optional parameter(s) to pass to target function. This parameter (s) will be stored to the "Return Value" of the called function. If there is more than one parameter, they will be stored to following lines, one parameter to one line, overwriting previous values. Line(s) with parameters name should be set to non-execute, otherwise the parameter(s) in the "Return Value" will be overwritten by a command after step is executed. Default: (empty)

Return value

Return value is defined by the [#return](#) command of the target function block.

6.1.1.2 #sync (Async function synchronization)

```
#sync | {<label>}
```

Synchronize to asynchronously called function in the caller thread. This function **blocks the execution flow** until specified async. function finishes.

Parameters

label	[string]	Label of function to be synchronized.
-------	----------	---------------------------------------

Return value

Return value can be defined by [#return](#) command of asynchronously called function.

6.1.1.3 #async (Async functions control)

```
#async | pause{: <thread0>}{; <thread1>}...{; threadN}
```

```
#async | resume{: <thread0>}{; <thread1>}...{; threadN}
```

```
#async | stop{: <thread0>}{; <thread1>}...{; threadN}
```

Pause, resume or stop thread(s).

- `pause` - if no `threadx` argument is passed, the function will pause *all other currently running* (including main) asynchronous functions (threads) except the calling function's thread. If at least one `threadx` argument is passed, the function will pause this/these specified thread(s).
- `resume` - if no `threadx` argument is passed, all asynchronous functions are resumed. If at least one `threadx` argument is passed, the function will resume this/these specified thread(s).
- `stop` - if no `threadx` argument is passed, all functions except the calling and main function are stopped. If at least one `threadx` argument is passed, the function will stop this/these specified thread(s). Using the stop method cannot be stopped the main or calling thread.

Parameters

<code>thread_x</code>	[string]	Name of the asynchronously called function to pause or resume.
---------------------------------	----------	--

There are two symbolic thread names:

- `@main` - identify the main thread (cannot be used during the stop operation)
- `@this` - identify the calling thread (cannot be used during the stop operation)

Return value

No return value.

```
#async | running?{: <thread>}
```

There are two possibilities:

- thread argument is **not** passed - get the count of currently called thread. Not depends if it is running or paused. The main thread is not counted.
- thread argument is passed - or the state of specified thread (running/paused)

Parameters

<code>thread</code>	[string]	Name of the function to get its state (paused/running). The <code>@main</code> symbolic thread name can be used.
---------------------	----------	--

Return value

- thread argument is **not** passed - return value is an integer number (if only main thread is running, the return value is 0)
- thread argument is passed - return value is 0 or 1, depends on state of the specified thread: paused = "0", running = "1"

6.1.1.4 #asyncflag (Async notification system)

Provides a mechanism to synchronize blocks within the asynchronously called functions using the flag/notification system. The flags are **global** for the running test file. They can be accessed from **any level** of testing thread.

By using the `lock` and `unlock` commands it is possible to get **unique access** to for example shared device, used by more threads. The `lock` method simply waits until the another thread will finish its work and unlock the specified flag.

```
#asyncflag | set| clear| get| lock| unlock: <name>
#asyncflag | wait: <name>{; change=[ enum] }{; timeout=[ int] }
```

Parameters

<code>set</code>	[enum]	Specifies the operation with the flag:
<code>clear</code>		<ul style="list-style-type: none"> • <code>set</code> - set the flag's value to logical 1 • <code>clear</code> - set the flag's value to logical zero • <code>get</code> - get the current flag's value • <code>wait</code> - block the execution until flag's value becomes a specified value, timeout parameter can be used here • <code>lock</code> - mark selected flag as locked, if the flag is already locked by another function, the program execution is blocked until the flag is unlocked • <code>unlock</code> - mark selected flag as unlocked (free the flag)
<code>get</code>		
<code>wait</code>		
<code>lock</code>		
<code>unlock</code>		
		If the required flag does not exist, it's automatically created with a default value. The default value is the opposite to that of waiting - this will cause block of the execution until specified flag is set/clear/changed.
<code>name</code>	[string]	The name of the flag.
<code>change</code>	[enum]	For the 'wait' operation only. Specifies for what the 'wait' operation will wait to continue the program execution: <ul style="list-style-type: none"> • <code>one</code> - wait for a change from logical 0 to logical 1 • <code>zero</code> - wait for a change from logical 1 to logical 0 • <code>any</code> - wait for any logical change
<code>timeout</code>	[int]	For the 'wait' operation only. Limit the time of waiting for specified change. In [ms].

Return value

No return value except the 'get' method, which returns '0' or '1'.

Note

When using `lock` and `unlock`, make sure, that the specified flag is always unlocked when operation finish. Otherwise the program will stop, because both (or more) threads will wait to flag unlock, which will never be done.

Examples

```
#asyncflag | set: done
```

Creates the "done"-named flag if does not exist and set its value to logical 1.

```
#asyncflag | get: done
```

Returns current state of the "done" flag, for example "1" in this case (after set).

```
#asyncflag | wait: done; change=zero; timeout=30000
```

Wait until "done" flag's value becomes zero (logical 0). This is done by using the 'clear' method in another execution thread. Wait for a maximum of 30 seconds.

6.2 Variables

Internal funTEST's variables can be used in the "**Parameters**" column (arguments of a command). All variables are replaced before executing a row. All variables names are **case-sensitive**.

Usage

The variable name is always between dollar signs: `$variable$`

Variables can be use anywhere in the text as many times as necessary.

Examples

```
#msg | "Hello $user-name$! Your login is $user-login$."
Display a simple message with currently logged operator name and login.
```

```
#file | text:write:"Date/time: $YYYY$-$MM$-$DD$ $hh$:$mm$:$ss$\nProject:
$project-name$ ($project-dir$)":file="c:\\loaded.txt"
Writes two lines using a file command with current date/time and currently loaded project name and its
directory.
```

6.2.1 Standard variables

List of predefined variables. They are automatically refreshed by funTEST.

Name	Description	Example value
\$project-dir\$	Directory of currently used project, without the ending backslash. All backslashes are doubled.	c:\\Users\\Public\\FPC\\funTEST\\projects\\ECU
\$project-name\$	Name of currently used project.	ECU
\$teststation-dir\$	Directory of currently used test-station, without the ending backslash. All backslashes are doubled.	c:\\Users\\Public\\FPC\\funTEST\\teststations
\$teststation-name\$	Name of currently used test-station.	MPT6
\$testfile-name\$	Name of currently loaded test-file.	Var1
\$user-login\$	Login name of currently logged-in user.	admin
\$user-name\$	Name of currently logged-in user.	Administrator
\$lang\$	Selected language, short 3-letter variant.	eng
\$YYYY\$	Current year, four-digit.	2015
\$YY\$	Current year, 00 to 99, last two-digit.	15
\$MM\$	Current month, 01 to 12, two-digit.	09
\$M\$	Current month, 1 to 12, single or two-digit.	9
\$DD\$	Current day, 01 to 31, two-digit.	02
\$D\$	Current day, 1 to 31, single or two-digit.	2
\$hh\$	Current hour, 00 to 23, two-digit.	08
\$h\$	Current hour, 0 to 23, single or two-digit.	8
\$mm\$	Current minute, 00 to 59, two-digit.	06
\$m\$	Current minute, 0 to 59, single or two-digit.	6
\$ss\$	Current second, 00 to 59, two-digit.	04
\$s\$	Current second, 0 to 59, single or two-digit.	4
\$panel\$	Active panel number, 0 to N. Set by #panel command.	3

6.2.2 Test-file variables

Before the row is executed, all test-file variables from the HEAD sheet are collected and added to the list of variables (replacing previous values).

Test-file variable names are always converted to lower-case.

If test-file localization is loaded, the caption of variable is translated.

6.2.3 Arrays

Since version 1.1.2104.1612, the funTEST supports array variables.

6.2.3.1 Definition

The place of definition is the same like any other variable - in the Variables section of HEAD sheet.

The name of variable is specific: it ends by "[]" (undefined/variable length) or "[n]" (fixed length).

In the test-file the value of array is stored in the specific variable's cell and values are separated by "|" (pipe) character.

Examples

`Array[]` - array variable "Index" with undefined count of array elements

`Array[15]` - array variable "Index" with fixed count of array elements

6.2.3.2 Length

Fixed length

The variable name is specified by "`name[length]`", where `length` is an integer number, higher than zero, which specified number of elements.

Any variable command is able to work with any element or a whole array without any initialization (e.g. read, rotate, ..). The index must be within the range of array. It's not possible to resize the array by writing outside the range of array.

Variable length

The variable name is specified by "`name[]`" - with just empty brackets. By default, there is no element in the array.

The array is resized automatically by writing to an element on any (non-existing) index. The reading is allowed only on existing index range. The whole-array operations are done over actual length of array.

6.2.3.3 Usage

Usage of array-types variables is the same like of non-array types, just use square bracket [n] at the end of the variable's name to specify the index of element. Commands to work with variables are standard [#cnt](#) and [#var](#).

It's possible to use direct indexing or indexing using other variable.

You can also access variable without any index specification (without "[..]"), the funTEST looks the variable like a single value with "|" separators.

Indexing

Array indexing is always zero-based: first element is at index 0.

Index offsets

You can also use offset while specifying index, the offset can also be a variable and can be negative (using "-")

sign) or positive ("+").

```
variable-name[ index{ +/-offset} ]
```

Examples

Array[2] - access element at index 2

Array[i] - access element at index, defined by variable "i", the contents of variable must be an integer number >= 0

Array[i+1] - access element at index, defined by variable "i", adding offset 1

Array[i-off] - access element at index, defined by variable "i", subtracting offset, defined by variable "off"

6.2.4 Commands

6.2.4.1 #cnt (Counter operations)

```
#cnt | <counter>
#cnt | <counter>=<value>
#cnt | <counter>+<offset>
#cnt | <counter>-<offset>
```

Set counter value, increment or decrement counter by a value or only read current counter value. Multiple counter operations can be done at once - use a semicolon ";" or doublecolon ":" to separate each counter operation. In this case, the result is the value of first operation.

Parameters

counter	[string]	Variable name to be used like a counter. Must be defined in test-file.
value	[number]	Integer value for variable.
offset	[number]	Integer value to be added to or subtracted from specified counter.

Return value

Result value of specified counter.

Examples

```
#cnt | ok
This will only return value of counter "ok".
```

```
#cnt | ok=0
Set "ok" counter value to 0.
```

```
#cnt | ok+3
Increment "ok" counter by 3.
```

```
#cnt | ok-1
Decrement "ok" counter by 1.
```

```
#cnt | pkg=25:ok=0:ng=0
Set "pkg" counter to 25 and "ok", "ng" counters to zero. Return value in this case is 25.
```

```
#cnt | pos[ 3]+1
Increment element at index 3 by +1 of array-variable "pos".
```

6.2.4.2 #var (Variable operations)

```
#var | s: <var_0>=<value_0>{: <var_1>=<value_1>}...{: <var_N>=<value_N>}
#var | st: <var_0>=<value_0>{: <var_1>=<value_1>}...{: <var_N>=<value_N>}
#var | a: <var_0>=<value_0>{: <var_1>=<value_1>}...{: <var_N>=<value_N>}
#var | r: <var_0>{: <var_1>}...{: <var_N>}
#var | c: <var_0>{: <var_1>}...{: <var_N>}
```

Operations with defines variables in the test-file:

- s - set variable(s) value(s)
- st - set variables(s) value(s), forced text
- a - append value(s) to specified variable(s)
- r - read values of specified variables
- c - clear values of specified variables

Variables shown on the operator's screen are updated automatically when changed on each test-program step.

Parameters

`varN` -or- `varN[index]` [string]

Variable name for single variables or variable name and index for arrays. Must be defined in test-file.

`valueN` [string]

String value to set (s, st) or append (a) to the variable or array's element at specified index.

By default, the OpenOffice will try to understand to the value like a number. There is a limitation in 15 valid digits, which OpenOffice can store (double precision number). If you need to force any value to a variable like a text, use the `st` (set text) command.

Return value

- s, st - value of the first passed variable
- a - final string of the first passed variable
- r - merged string of variables to read
- c - no return value

Examples

```
#var | s: text="abc"
```

Set value of "text" variable to "abc", return value will be "abc"

```
#var | s: text="abc": str="def"
```

Multiple variable set at once - variable "text" to "abc" and var "str" to "def". Return value will be "abc"

```
#var | r: text: str
```

Read variables "text" and "str", return will be for example "abcdef" (according to previous example call)

```
#var | c: text: str: arr[5]
```

Clear "text" and "str" variables and element with index 5 of "arr" array variable.

```
#var | a: text="123"
```

Append the string "123" to the variable "text"

```
#var | s: arr[5]="123"
```

Set the element with index 5 of array-variable "arr" to "123"

```
#var | s: arr[ i] =" abc"
```

Set variable element at index, defined by variable "i" to "abc".

```
#var | arr: ror: <var_0>{: <var_1>: . . . : <var_N>}
```

```
#var | arr: rol: <var_0>{: <var_1>: . . . : <var_N>}
```

Rotate elements in array variable to right (ror) or left (rol). The length of array remains the same.

Parameters

var	[string]	Variable(s) to rotate, the variable must be an array-type.
-----	----------	--

Return value

No return value

Example

For example, let's have a simple array variable with name "fields":

1a	2b	3c	4d
----	----	----	----

```
#var | arr: ror: fields
```

Rotate elements to right, by 1:

4d	1a	2b	3c
----	----	----	----

```
#var | arr: rol: fields
```

Rotate elements to left, by 1:

2b	3c	4d	1a
----	----	----	----

```
#var | arr: ror: array1: array2
```

Rotate two arrays at once

```
#var | arr: set: <var>: <value>{: size=[ int] }
```

Rotate elements in array variable to right (ror) or left (rol). The length of array remains the same.

Parameters

var	[string]	Variable to fill by a value, the variable must be an array-type.
size	[int]	Optional size, when the target variable is variable-length type. For fixed-length arrays the argument is ignored.
value	[string]	Value to fill the array.

Return value

No return value

Example

```
#var | arr: set: numbers: "0"
```

Set all elements of numbers array to "0".

6.3 Interrupt events

The test-file supports events, automatically raised by funTEST while specified action occurs. These events are defined like a sub-programs with special label names and format:

```
*event( event-type{ : args} )
```

event-type	[string]	Event type to catch, see following section for supported events.
args	[string]	Optional arguments to specific event types.

Examples

```
*event( prj-closed)
```

Catch the project-closing event, no arguments.

```
*event( run-every: 15m)
```

Setup the event, which will run automatically every 15 minutes (defined by "15m" argument).

Event sub-programs acts like standard asynchronously called methods (by funTEST), they can have passed some argument and must have a #return. **Make sure that defined event is outside the main program loop. Recommended location is at the end of the test-file.**

6.3.1 Event types

6.3.1.1 prj-closing (Project closing)

```
prj-closing
```

Occurs before the test-file is closed. The closing can be refused by the #return argument.

Return argument

0 = refuse closing of test, continue normally

(otherwise) = finish closing

6.3.1.2 prj-closed (Project closed)

```
prj-closed
```

Occurs after test-file has been closed. Any GUI (Operator interface) command is denied here.

6.3.1.3 oper-changed (Operator changed)

```
oper-changed
```

Occurs after a new user is logged in (when another user is already logged in).

Return value

New logged-in user name will appears in the "Return value" column.

6.3.1.4 run-at (Run at specified time)

```
run-at: <hour>{: <min>{: <sec>}})
```

This label will be called by funTEST at specified time in a day. Call is repeated every day at this time. The time can be specified by hour only or also by minutes and seconds.

Parameters

hour	[int]	Hour to run, 0 .. 23
min	[int]	Min to run, 0..59 Default: 0
sec	[int]	Sec to run, 0..59 Default: 0

Examples

```
*event( run-at: 1)
Run at 01:00.
```

```
*event( run-at: 13: 30)
Run at 13:30.
```

```
*event( run-at: 15: 10: 30)
Run at 15:10:30.
```

6.3.1.5 run-every (Run every interval)

```
run-every: <ms>{ : <sec>{ : <min>{ : <hour> } } } )
```

This label will be called by funTEST repeatedly, after specified interval elapses. Time is defined by a sum of all parameters. It is possible to define the interval for example by only one parameter - passing i.e. 150 seconds is equivalent to 2 minutes and 30 seconds and so on. The next call is ignored when previous call is not finished.

Parameters

ms	[int]	hundreths of ms (1 = 100ms)
sec	[int]	seconds
min	[int]	minutes
hour	[int]	hours

Examples

```
*event( run-every: 5)
Run every 0,5 second.
```

```
*event( run-every: 0: 30)
Run every 30 second.
```

```
*event( run-every: 0: 90)
Run every 1,5 minute.
```

```
*event( run-every: 0: 30: 1)
Run every 1,5 minute.
```

```
*event( run-every: 0: 0: 60)
Run every 1 hour.
```

```
*event( run-every: 0: 0: 0: 1)
The same 1 hour, but using alternative declaration.
```

```
run-every: <interval>
```

The same functionality, but using simplified interval declaration.

Parameters

interval	[string]	One of following notations can be used: <ul style="list-style-type: none"> • <i>N</i>ms - number of milliseconds • <i>N</i>s - number of seconds • <i>N</i>m - number of minutes • <i>N</i>h - number of hours
----------	----------	--

..where "N" is an integer number and "ms/s/m/h" is suffix

Examples

```
*event( run-every: 10s)
Run every 10 second.
```

```
*event( run-every: 15m)
Run every 15 minute.
```

```
*event( run-every: 60m)
Run every 1 hour.
```

```
*event( run-every: 1h)
Run every 1 hour, but using alternative declaration.
```

6.4 Commands

6.4.1 Syntax

FunTEST has a set of dedicated commands to control the program flow.

In this chapter the specified syntax of command usage descriptions and data-types will be described.

Basics

Every command is divided into two parts:

- main command - column "Command (Device)" in the test-file
- parameters - column "Parameters"

In this manual, the main command and parameters are separated by a pipe-character - "|".

Example:

```
#retclear | from=@this+1;to=@this+10
```

...where "#retclear" is the main command and "from=@this+1;to=@this+10" are parameters.

Syntax

```
#command | sub-command: required=[ type] { ; optional=[ type] } ;
    <required-value-only> { ; <optional-value-only> }
```

Symbols:

- < > - value-only parameter
- { } - optional
- [] - data type

Description:

#command - main command

sub-command - optional sub-command, must be separated by a colon (":") from parameters

required - required named parameter

optional - optional named parameter

<required-value-only> - required value-only parameter

<optional-value-only> - optional value-only parameter

[type] - parameter value type, see section "Value types" below

Example, minimum length:

```
#command | sub-command: required="abc"; "123"
```

Example, maximum length:

```
#command | sub-command: required="abc"; optional="def"; "123"; "456"
```

Parameters

Command parameters must be also written in a specific format.

It distinguishes several types:

- *sub-command*
For example: "panel: 0", where the "panel" is a sub-command. Sub-command (if present) must always be at first place in the sequence and no more than once. If any named or value-only parameter follows, they are separated by a colon from sub-command.
- *named parameter* (value name + its value)
For example: "from=@this+1; to=@this+10", where "from" and "to" are names of parameters and @this+1 and @this+10 are parameter values. Parameters are separated by a semicolon.
- *value-only parameter*
For example: "abc;123; "d e f"". Value-only parameters are also separated by a semicolon.

Parameter types can be combined. In the parameters sequence can be up to one sub-command and unlimited number of named and value-only parameters.

For example: set: value=5; "abc"; "def" ("set" is sub-command, "value=5" is named parameter and "abc" and "def" are value-only parameters).

Parameter value format:

Parameter value can be passed using quotes. Quotes are recommended when spaces in the value are present. They are required when you need to use special characters in the parameter value like colon or semicolon, which are reserved for separating parameters.

For example, to pass a value containing a semicolon-separated text values, you have to type:

```
set: value="abc; def; ghi"
```

Escape sequences

When you need to pass for example a non-ASCII character, quote or a back-slash character, you have to use *escape sequences*. Escape sequence is a sub-string begins with the back-slash character (\) following a specified number of characters. Escape sequence can substitute any ASCII or non-ASCII value.

Example:

```
set: value="\ "abc\"; \\def\\; ghi\r\n"
```

When you split the value parameter text by semicolons, you will get following three five-character long strings:

- "abc"
- \def\
- ghi + ASCII character 13 (CR) + ASCII character 10 (LF)

There is no other way how to pass these characters without using escape sequences.


Supported sequences:

- \ " - quote character
- \\ - back-slash character
- \r - carriage return (CR) character, ASCII 13, mostly used like line-ending character

- \n - line feed (LF) character, ASCII 10, mostly used like line-ending character
- \xYY - where YY is a hexadecimal notation 00 to FF, it can represent any character value, case insensitive

Value types

Parameter value (of named or value-only parameter) is always a plain text. Many commands require this text special-formatted to represent for example an integer number, boolean value and etc. See all value types in the table below.

Value type	Valid values	Description
[string]	(text)	Plain text, "funTEST system" for example.
[number]	-2147483648 to 2147483647	32-bit integer signed number with specified range.
[bool]	<ul style="list-style-type: none"> • False = "0", "no", "false" • True = "1", "yes", "true" 	Boolean (logical) value.
[enum]	List of specific values	An enumeration, it is almost the same like [string], but only specified values are allowed. For example, defined values of enumeration are "one", "two" and "three" - that means this enum-type parameter can contain only one of these 3 options)
[list]	val ₁ , val ₂ , val ₃ , ..., val _N	Simple comma-separated value list ("abc,123,+/-" for example).
[color]	<ul style="list-style-type: none"> • RGB • RRGGBB • {dark- dark-dark- light- light-light-} <color-name> -or- gray-XX 	<p>Color notation. HTML-like style without a sharp (#) character. The value can be a 3-character (format "RGB"), 6-characters length (format "RRGGBB") or color name (see the list below).</p> <ul style="list-style-type: none"> • "RGB" short-format: R, G and B are hexadecimal notations in range from "0" to "F", case insensitive (for example: "000" = black, "FFF" = white, "0F0" = green, ...) • "RRGGBB" full-format: RR, GG and BB are hexadecimal notations in rage from "00" to "FF", case insensitive (for example "000000" = black, "FFFFFF" = white, "00FF00" = green, ...) • Color-name There are 10 standard named colors: black, white, red, green, blue, yellow, cyan, magenta, orange and gray Any color except "black" and "white" can be <i>darkened</i> or <i>lighten</i> by prefixes: "light-" -or- "light-light-" (light/more light) "dark-" -or- "dark-dark-" (dark/more dark) Gray level can also be defined by format "Gray-XX", where XX is the level of lightness in % in range from 01 (almost black) to 99 (almost white) 

[time]

- s
- s.f
- m:ss
- m:ss.f

Time notation, defined formats:

- Seconds only: "s", where "s" are seconds (number from 0 to 59, one or two-digit)
- Seconds with tenth fraction: "s.f", where "f" are tenth of second - number from 0 to 9. For example: "37.8"
- Minutes and seconds: "m:ss", where "m" are minutes (number from 0 to 59) and "ss" seconds (must be two-digit). For example: "12:06"
- Minutes, seconds and tenth of seconds: "m:ss.f", for example: "8:09.7"

6.4.2 Summary

Alphabetical sorted list of all available commands.

- [#adjust](#)
- [#async](#)
- [#asyncflag](#)
- [#bct](#)
- [#call](#)
- [#callasync](#)
- [#catch](#)
- [#catchio](#)
- [#cellcopy](#)
- [#cellerase](#)
- [#cellread](#)
- [#cellwrite](#)
- [#cnt](#)
- [#dlg](#)
- [#dummy](#)
- [#export](#)
- [#extprocess](#)
- [#file](#)
- [#get](#)
- [#goto](#)
- [#local](#)
- [#login](#)
- [#msg](#)
- [#onerror](#)
- [#onfail](#)
- [#oninput](#)
- [#panel](#)
- [#print](#)
- [#resultlist](#)
- [#retclear](#)
- [#return](#)
- [#stat](#)
- [#statrec](#)
- [#statsave](#)
- [#status](#)
- [#stopwatch](#)
- [#str](#)
- [#sync](#)
- [#testfile](#)
- [#throw](#)
- [#userbtn](#)
- [#var](#)

- [IO commands](#)
- [MX commands](#)

6.4.3 Flow control

6.4.3.1 #goto (Jump to label)

```
#goto | <label-name>
```

Jump to a target label.

Parameters

label-name	[string]	Target label name to jump to. There is a special predefined label <code>@this</code> which represent current row. This can be used to make the infinite main loop without defining any other label.
------------	----------	--

Return value

No return value.

6.4.3.2 #call (Call a function)

```
#call | <function-name>{:<param0>:...:<paramN>}
```

Call a function. Function is a block of code begins by the label and ends by the [#return](#) (or [#throw](#)) command. Typically the block should be placed outside the main program loop. The `#call` blocks the executing until sub-routine is finished. The return value (`#return` of sub-routine) is written to Return Value.

Starting funTEST version 1.0.1906.311 multiple parameters are supported. Previous versions support `param0` only.

Parameters

function-name	[string]	Function name to call
param	[string]	Optional parameter(s) to pass to target function. This parameter (s) will be stored to the "Return Value" of the called function. If there is more than one parameter, they will be stored to following lines, one parameter to one line, overwriting previous values. Line(s) with parameters name should be set to non-execute, otherwise the parameter(s) in the "Return Value" will be overwritten by a command after step is executed. Default: (empty)

Return value

Return value is defined by the [#return](#) command of the target function block.

6.4.3.3 #return (Return from function)

```
#return | {<return-value>}
```

This command represents the end of the function block.

Parameters

return-value	[string]	Optional parameter, function can return a blank value. Otherwise this is any value to be returned by a function block. Default: (empty)
--------------	----------	---

Return value

No return value.

6.4.3.4 #throw (Exit function with error)

Request funTEST version: 1.0.1906.311

```
#throw | { <message> }
```

Return from function with an error. This will cause ReturnStatus = 1 at the place of function calling (using #call). If the async-called function is existed by #throw, the error is passed to #sync function. It is possible to return custom error message or pass last caught error by #onerror functionality.

Parameters

message	[string]	Optional error message to return from the function. If no message is passed, the #throw command will return last error, caught by #onerror functionality.
---------	----------	---

Return value

No return value.

Examples

```
#throw | "Barcode reader error!"
Exit the function with custom error message.
```

```
#throw
Exit the function with last error, caught from previously set #onerror.
```

6.4.3.5 #onerror, #onfail (Check for error or fail)

These commands enables to check for an error (means Return Status <> 0) or fail (means Judge is not empty and <> 0).

Note:

On-fail action is not executed when the step ends with an error. If you want to treat both situation, you have to define behaviour for both on-fail and on-error (can be the same).

It is possible to:

- just set a number of retries of following step(s)
- perform a function call
- go to to a specified label

```
#onerror / #onfail | none
```

Disable the error or fail checking functionality.

```
#onerror / #onfail | retry: <count>
```

Set number of retries. The funTEST will repeat the line, where an error or fail occurs several times before proceeding.

Parameters

count	[int]	Set the number of repeats. Must be a positive number. "0" means no repeats.
-------	-------	---

Return value

No return value.

Examples

```
#onerror | retry:3
```

Set the number of retries to 3. That means - if the first will fail, it will be repeated by 3 times (total of 4 calls).

```
#onerror / #onfail | <command>: <label>{: retry=[ int]}
```

Set number of retries. The funTEST will repeat the line, where an error or fail occurs several times before proceeding.

Parameters

command	[enum]	What should be done if error or fail has occurred. It can be on of these option: <ul style="list-style-type: none"> • goto - go to line with a specified label, no return • call - call specified function block, program will return at the place where an error or fail occurred
label	[string]	Target label to go to or name of function block to call. Function block must ends with #return command.
retry	[int]	Optionally the number of retries can be set before go to a label or call a function block.

Default: 0

Return value

No return value.

Examples

```
#onerror | goto: error
```

If an error occurs, the funTEST go to a line labeled "error".

```
#onerror | goto: error: retry=3
```

If an error occurs the current line is repeated by a maximum of 3 times. If not success, then funTEST go to a line labeled "error".

```
#onerror | call: error_func: retry=3
```

Almost the same like above. The difference is, that the function is called there and program will continue from the line where an error has occurred.

6.4.4 Operator interface

6.4.4.1 #msg (Show a message)

```
#msg | { <text> } { ; type=[ enum| string ] } { ; size=[ enum ] } { ; image=[ string ] }
      { ; color=[ color ] } { ; bg=[ color ] } { ; tsize=[ int ] } { ; tpos=[ enum ] }
```

Show a formatted message on operator screen.

Parameters

text	[string]	Text to display. If test-file localization is loaded, the text is automatically translated.
type	[enum string]	Message type and text position if both displayed.

		<ul style="list-style-type: none"> • <code>auto</code> - determine the type based on 'text' and 'image' arguments ('text' only = type text, 'image' only = type image, both 'text' and 'image' = both) <p>-or-</p> <ul style="list-style-type: none"> • <code>im</code> or <code>image</code> - show image • <code>t</code> or <code>text</code> - show text in default position • <code>tt</code> - show text on the top • <code>tb</code> - show text on the bottom • <code>tl</code> - show text on the left side • <code>tr</code> - show text on the right side <p>Type is a combination of 1 - 2 type string above, using plus '+' character, i.e.: "im" (image only), "im+t" (image and text), "im+tt" (image and text on top)</p> <p>Default: <code>auto</code></p>
<p>size</p>	<p>[enum]</p>	<p>Message size (area of operator screen):</p> <ul style="list-style-type: none"> • <code>tiny</code> - small box for 1-2 lines of text • <code>medium</code> - medium-sized box, about one-half of the operator screen • <code>large</code> - full-screen message on operator screen (excluding top and right-side menus) <p>Default: <code>tiny</code></p>
<p>image</p>	<p>[string]</p>	<p>Image to display. Required when type is specified to display an image. Supported image formats: JPEG, PNG, GIF and BMP</p> <ul style="list-style-type: none"> • path to file on a hard-drive (i.e. "c:\path\to\image.png") • device plug-ins from version 2.0: "plugin://<device-alias>" <p>When image source is set to plug-in, the plug-in can render custom image to the operator's screen independently.</p> <p>Default: (none)</p>
<p>color</p>	<p>[color]</p>	<p>Color of message text. Default: <code>black</code></p>
<p>bg</p>	<p>[color]</p>	<p>Color of message background. Default: <code>white</code></p>
<p>tsize</p>	<p>[int]</p>	<p>Text size in points. Default: 24</p>
<p>tpos</p>	<p>[enum]</p>	<p>Possibility to specify position of the text when 'type' argument is not used or set to 'auto' and both image and text are displayed. Valid positions:</p> <ul style="list-style-type: none"> • <code>top</code> (above the image) • <code>bottom</code> (below the image) • <code>left</code> • <code>right</code> <p>If the 'type' argument is specified, this argument is ignored. Default: <code>bottom</code></p>

Return value

No return value.

Examples

```
#msg | "Simple message to show..."
```

Display a simple tiny message with text specified above.

```
#msg | "Message to show...";size=medium;tsize=40;image="c:\\path\\to\\image1.jpg";color=light-red
```

Display medium-sized message with a light-red colored text and image specified by path and text below of image. Set the text-size to 40pt (default is 24).

```
#msg | size=large;image="c:\\path\\to\\image2.png";bg=black
```

Display a full-screen image only with black background.

```
#msg | "Text left of the image";size=medium;tpos=left;image="c:\\path\\to\\image2.png"
```

Display a medium-sized message with text aligned to left side.

```
#msg | "Text left of the image";size=medium;type="im+t1";image="c:\\path\\to\\image2.png"
```

Display a medium-sized message with text aligned to left side (an alternative to writing above using exact specification by 'type' argument).

```
#msg | size=large;image="plugin://Remote"
```

Display a full-screen image and allow the plug-in with alias "Remote" to send the image directly to operator's interface.

```
#msg | push
```

Save last shown message (including all parameters) to memory. This commands requires any previously shown message. The memory is shared across all threads - the message can be stored in one thread and restored in another thread.

```
#msg | pop
```

Restore last saved message from the memory. This commands requires previously saved message using push. Calling this command will not delete the saved message, so it's possible to call it repeatedly.

The push and pop commands gives the opportunity to the programmer to temporary show other message and the easily restore the previous message.

6.4.4.2 #dlg (Show a dialog)

```
#dlg | <text>{;type=[ enum| string] }{;w=[ number] }{;h=[ number] }{;image=[ string] }
    {;color=[ color] }{;bg=[ color] }{;tsize=[ number] }{;items=[ list] }
    {;edit=[ bool] }{;input-text=[ string] }{;input-mask=[ bool] }{;buttons=[ list] }
    {;tpos=[ enum] }{;iomap=[ list] }
```

Show an overlay dialog with formatted message on operator screen. This function blocks executing the program until one of dialog button is pressed or item is selected.

Parameters

text	[string]	Text to display. If test-file localization is loaded, the text is automatically translated.
------	----------	--

type	[enum string]	<p>Message type and text position if both displayed:</p> <ul style="list-style-type: none"> • <code>auto</code> - determine the type based on 'text' and 'image' arguments ('text' only = type text, 'image' only = type image, both 'text' and 'image' = both) <p>-or-</p> <ul style="list-style-type: none"> • <code>im</code> or <code>image</code> - show image • <code>t</code> or <code>text</code> - show text in default position • <code>tt</code> - show text on the top • <code>tb</code> - show text on the bottom • <code>tl</code> - show text on the left side • <code>tr</code> - show text on the right side <p>Type is a combination of 1 - 2 type string above, using plus '+' character, i.e.: "<code>im</code>" (image only), "<code>im+t</code>" (image and text), "<code>im+tt</code>" (image and text on top)</p> <p>Default: <code>auto</code></p>
w	[number]	<p>Dialog width, percent of operator screen. Range: 20 - 100 Default: 80</p>
h	[number]	<p>Dialog height, percent of operator screen. Range: 20 - 100 Default: 80</p>
image	[string]	<p>Image to display. Required when type is specified to display an image. Supported image formats: JPEG, PNG, GIF and BMP</p> <ul style="list-style-type: none"> • path to file on a hard-drive (i.e. "<code>c:\path\to\image.png</code>") • device plug-ins from version 2.0: "<code>plugin://<device-alias></code>" <p>When image source is set to plug-in, the plug-in can render custom image to the operator's screen independently.</p> <p>Default: (none)</p>
color	[color]	<p>Color of message text. Default: <code>black</code></p>
bg	[color]	<p>Color of message background. Default: <code>white</code></p>
tsize	[number]	<p>Text size in points. Default: 24</p>
tpos	[enum]	<p>Possibility to specify position of the text when 'type' argument is not used or set to 'auto' and both image and text are displayed. Valid positions:</p> <ul style="list-style-type: none"> • <code>top</code> (above the image) • <code>bottom</code> (below the image) • <code>left</code> • <code>right</code> <p>If the 'type' argument is specified, this argument is ignored.</p> <p>Default: <code>bottom</code></p>
items	[list]	<p>A comma-separated list of text items. One of them can be selected in the dialog.</p>

		When no buttons, text and image is passed, the result is the dialog with only the list of items to select. Default: (none)
edit	[bool]	If true, the edit-box is shown on the dialog. Default: false
input-text	[string]	The default text to be prepared in the edit-box.
input-mask	[bool]	Mask the text by a standard system password character. Default: false
buttons	[list]	A comma-separated list of buttons to show. Possible values are: <ul style="list-style-type: none"> • ok • cancel • yes • no • retry If edit box is shown or item values to select are passed, the "ok" button should be shown, because it confirms the typed text. Default: ok (if no items and iomap are passed, otherwise no button is shown)
iomap	[list]	Mapping buttons to specified IO (inputs only) aliases from teststation's IO mapping system . Comma-separated list in following possible formats: <ul style="list-style-type: none"> • Inputs aliases only: $in_{button(0)}, in_{button(1)}, \dots, in_{button(N)}$ (buttons to pair depends equals to order in the buttons argument) -or- • Specified button(s) to specified alias(es): $io_0 > button_0, in_1 > button_1, \dots, in_N > button_N$ (buttons are paired to inputs directly) -or- • Combination of both formats: $in_{button(0)}, in_{button(1)}, io_2 > button_2, \dots, in_N > button_N$ (first two buttons are paired by buttons argument position and others are directly paired) Default: (none)

Return value

If no items are passed to the dialog and no edit-box is shown, the return value is the name of button pressed with a "@" character at first place ("@ok", "@cancel", "@retry", etc..). Otherwise, the return value is the selected item or user-entered text.

Examples

```
#dlg | "Simple dialog to confirm.."
```

Show a simple confirmation dialog with one button - "OK". Return value will be always "@ok".

```
#dlg | "Select an item";w=100;h=100;items="one,two,three,four,five"
```

Show a full-screen dialog (width and height is 100%) to select one of predefined items. Return value will be one of the items.

```
#dlg | "Enter your name: ";w=70;h=50;edit=true
```

Show a dialog with an edit-box. Return value will be the entered text.

```
#dlg | "Do you want to continue testing?";buttons=no,yes
```

Show a dialog with a simple text and "No" and "Yes" button. The order of buttons in the bottom-right

corner will be in order - "No", "Yes".

```
#dlg | items="First, Second, Third"
```

Show a dialog with only list of items to choose. No prompt, image and button bar is displayed.

```
#dlg | "IO paired dialog"; buttons="ok, cancel"; iomap="start, stop"
```

Show a dialog with two buttons and map OK button to "start" IO alias and Cancel button to "stop" alias

```
#dlg | "IO paired dialog"; buttons="ok, cancel"; iomap="start, stop, reject>no"
```

Show a dialog with two buttons and map OK button to "start" IO alias, Cancel button to "stop" alias and (not shown) No button to "reject" alias

```
#dlg | "Only IO can confirm..."; iomap="start>ok, stop>cancel"
```

Show a dialog with a text and buttons. Only specified IO inputs can confirm the dialog, in this case "start" alias works like OK button and "stop" alias works like Cancel button.

6.4.4.3 #status (Set test status of current panel)

```
#status | {<status>}{: panel=[ list] }{: color=[ color] }
```

Control status label of current or specified panel(s).

Parameters

status	[string]	Status custom text of current panel or one of predefined values below: <ul style="list-style-type: none"> • @ready - Ready/In queue • @pass - Test passed • @fail - Test failed • @test - Testing in progress Default: no text
panel	[list]	Specify panel(s) to change status label at once. Format: comma separated panel numbers or asterisk (*) to select all existing panels. Default: current panel
color	[color]	Background color of specified panel when the custom text is used. The "@ready/pass/fail/test" will have always fixed colors (gray/green/red/yellow). Default: light-yellow

Return value

No return value.

Examples

```
#status | @pass
```

Set status of current panel to "passed".

```
#status | @ready: panel=*
```

Set "ready" status for all panels.

```
#status | @test: panel=1, 3, 4
```

Set "test" status for selected panels.

```
#status | "Processing...": color=light-blue
```

Set custom status text of current panel to "Processing..." with light blue background color.

```
#status | panel=4: color=blue
Set panel's 4 color to blue, without any text.
```

6.4.4.4 #resultlist (Result list operations)

```
#resultlist | clear{: panel=[ int]}
```

Clear all results or if panel parameter specified, clear selected panel's results only.

Parameters

panel [number] Specify of which panel's results will be cleared.

Return value

No return value.

```
#resultlist | showerrors
#resultlist | showall
```

Show only error results or all results.

Parameters

No parameters.

Return value

No return value.

```
#resultlist | panel:<panel>
```

This command allows to set current panel number. Valid numbers must be defined in the test-program.

Parameters

panel [number] or [enum] An integer number specifying panel number or one of following commands:

- **all** - show all panels in the result list
- **current** - show currently selected panel in the result list

Return value

No return value.

```
#resultlist | reqconfirm
```

Shows a confirm button on the testing screen. This function blocks the executing of program until user confirms results.

Parameters

No parameters.

Return value

No return value.

```
#resultlist | sort:<type>
```

Sorts current view of result-list. The sorting is not done on-the-fly and must be called manually. Sorting cannot be reverted, once it's done the previous order cannot be restored.

Parameters

`type` [enum] Type of sorting, current only one is available:
 • `FailsOnTop` - moves FAILs to the top of the list

Return value

No return value.

6.4.4.5 #panel (Panels control)

```
#panel | {<panel>}
```

This command allows to set current panel number. Valid numbers must be defined in the test-program.

Parameters

`panel` [number] An integer number specifying the current panel number.
 If not passed, current panel is not changed. Only currently selected panel will be returned.

Return value

Number of currently selected panel.

```
#panel | set{:size=[ string]}{:xsize=[ number]}{:ysize=[ number]}
        {:numbering=[ string]}{:xdef=[ number]}{:ydef=[ number]}
```

Programmatically change the panel configuration while test is running.

Parameters

`size` [string] Target panel size in format "<columns>x<rows>", e.g. "5x2"

Default: load from test-file HEAD definition

`xsize, ysize` [number] Target panel size in separated arguments.

Default: load from test-file HEAD definition

`numbering` [string] Target panel numbering:

- Custom numbers format: columns separated by "," and rows by ";"
 (e.g. "1, 3, 2; 4, 6, 5" = 2 rows/3 columns)

The number of rows and columns must match to size (or `xsize/ysize`) argument.

- `byrow`: auto-numbering by rows

1	2	3
4	5	6
7	8	9

- `bycol`: auto-numbering by columns

1	4	7
2	5	8
3	6	9

Default: load from PANEL definition

`xdef, ydef` [number] Load definition from PANEL sheet at `xdef/ydef` offset.

Default: load from PANEL definition

Return value

Number of currently selected panel.

6.4.4.6 #stopwatch (Integrated stopwatch control)

Using this command you can show/hide/start/stop/reset and set parameters of integrated stop-watch system. Stop-watch can be shown on the operator screen, or run in the background to measure the testing cycle time.

```
#stopwatch | show
#stopwatch | hide
```

Show or hide the stop-watch on the operator's screen.

Parameters

No parameters.

Return value

No return value.

```
#stopwatch | start:{show=[bool]}
#stopwatch | stop:{hide=[bool]}
#stopwatch | reset
```

Run, stop or reset stopwatch. Optionally the stop-watch can be automatically show/hide on the operator screen by using start/stop commands. Reset command sets the default time of stopwatch - to zero in normal mode and to time limit in the countdown mode.

Parameters

show	[bool]	If true, stopwatch are automatically displayed on the operator screen using the start command. Default: false
hide	[bool]	If true, stopwatch are automatically hidden using the stop command.

Return value

No return value.

```
#stopwatch | set:{fmt=[string]};{cntdown=[bool]};{limit=[time]}
```

Set parameters of stop-watch like operator screen display format, mode and time-limit.

Parameters

fmt	[string]	Set the time display format of stopwatch on the operator screen. You can build your own style using these basic custom format specifiers below: <ul style="list-style-type: none"> • h, hh - hour without or with the leading zero if one-digit only • m, mm - minute without or with the leading zero • f, ff, fff - tenths, hundredths of second or milliseconds, always with leading zero(s)
-----	----------	--

Delimiter characters like dot '.', color ':' and others must be escaped using the backslash character '\'. It is possible to include any custom string using the literal delimiter -

apostrophe (for example 'min') at any place of the string. Any other unescaped character, or character not between ' is interpreted as a custom format specifier.

The default format can be set by "default" keyword.

cntdown	[bool]	<p>Default: "m' min, 'ss' sec'" ("0 min, 00 sec")</p> <p>Mode of stop-watch:</p> <ul style="list-style-type: none"> • false - normal mode (upcounting) • true - countdown mode <p>To change stop-watch mode is not possible when running.</p>
limit	[time]	<p>Default: false</p> <p>The time limit. This is only for operator screen. If an overtime occurred, time stopwatch color on the operator screen becomes red.</p> <p>No limit can be set by "none" keyword.</p> <p>Default: none (no time limit)</p>

Return value

No return value.

Examples

```
#stopwatch | set:fmt="' - 'm' min, 'ss\\.f' sec' ";cntdown=true;limit="1:15"
Set display format style to "- x min, yy.z sec" (for example "- 0 min, 56.2 sec"), countdown mode and time limit to 1 minute and 15 seconds.
```

```
#stopwatch | set:fmt=default;cntdown=false;limit=none
Restore default display format, set normal mode (upcounting) and no time limit.
```

```
#stopwatch | time?:{<value-type>}
#stopwatch | time?:{fmt=[string]}
```

Read current value of stop-watch.

Parameters

value-type	[enum]	<p>Get current time of stopwatch in one of following unit:</p> <ul style="list-style-type: none"> • msec -or- ms: total number of milliseconds (integer number) • sec -or- s: total number of seconds (decimal) • min -or- m: total number of minutes (decimal) • hour -or- h: total number of hours (decimal)
fmt	[string]	<p>Get current time of stopwatch in specified format. The formatting string is almost the same as for the "set" command (except the default keyword).</p> <p>Short summary:</p> <ul style="list-style-type: none"> • h, hh - hours • m, mm - minutes • s, ss - seconds • f, ff, fff - 1/10, 1/100 or 1/1000 of second <p>It is possible to get full time representation or for example only second or minute part of time and so on.</p>

<code>limit</code>	<code>[time]</code>	<p>Default: "mm\:ss\.f" (for example "02:31.7" = 2 minutes, 31.7 seconds)</p> <p>The time limit. This is only for operator screen. If an overtime occurred, time stopwatch color on the operator screen becomes red.</p> <p>No limit can be set by "none" keyword.</p> <p>Default: none (no time limit)</p>
--------------------	---------------------	---

Return value

Return value depends on variant of a command used - means if `<value-type>` or `fmt` argument is used.

- a) `value-type` used - return value is *integer or decimal*, total number of msec -or- sec -or- min -or- hour, decimal notation if needed
- b) `fmt` used - return value is a time in specified format, by default in "mm: ss. f" format (minutes, seconds and tenth of seconds)

Only one argument can be used at the same time. If both, the `value-type` has a priority. If none, the `fmt` default format is used.

If the countdown is active and the time is below zero (= over limit), the value starts by negative "-" sign.

Examples

The following examples expecting current time of stopwatch for example "2: 25. 3" (2 min, 25 sec, 300 msec).

```
#stopwatch | time?
```

Get current time in the default format. Return value will be: "2: 25. 3"

```
#stopwatch | time?:sec
```

Get total number of seconds. Return value will be: $2*60+25+0.3 = "145. 3"$

```
#stopwatch | time?:fmt="ss"
```

Get the second part of time, tow-digit format. Return value will be: "25"

6.4.4.7 #userbtn (User-button control)

This command enables to add/modify/remove the user button(s) on operator's interface sidebar. Click the button performs the asynchronous function call to the specified label.

```
#userbtn | add: <id>; call=[ string] { ; param=[ string] } { ; caption=[ string] }
           { ; enabled=[ bool] } { ; image=[ path] }
```

Add a new user-button with a specified ID and parameters.

```
#userbtn | set: <id_0>{ ; <id_1> } ... { ; <id_N> } { ; param=[ string] } { ; caption=[ string] }
           { ; enabled=[ bool] } { ; image=[ path] }
```

Modify parameter(s) of the existing user-button with specified ID. If the ID is *, the parameters are passed to the all buttons.

```
#userbtn | remove: <id_0>{ ; <id_1> } ... { ; <id_N> }
```

Remove the existing user-button with specified ID. Use * as ID to remove all existing buttons.

Parameters

<code>id -or- id_x</code>	[string]	The unique ID(s) of the button(s). ID is used to identify the button in other functions. An asterisk (*) used like an ID identifies all existing buttons - this cannot be used when adding buttons.
<code>call</code>	[string]	Target label of the function which will be asynchronously called by clicking the specified button.
<code>param</code>	[string]	Optional parameter to pass to the target function. If exists, the value will be written in to the "Return Value" column on the line, where the function label is declared.
<code>caption</code>	[string]	Caption of the button. This parameter is not required, but recommended.
<code>enabled</code>	[bool]	If true, the button is disabled, if false the button is enabled. Default: false
<code>image</code>	[path]	Picture of the button. This parameter is not required, but recommended. Source picture can be in the PNG/JPEG/BMP/GIF format, highly-recommended is the PNG format with the resolution of 64x64 pixels.

Return value

No return value.

Notes

Once the function is called, it is not possible to call it again until it finishes, otherwise an error message is thrown. It's recommended to disable the button using the `set:<id>;enabled=false` command immediately after asynchronous function is called and enable the button at the end of the function. The error message is shown also in the case, that the target label does not exist

Examples

```
#userbtn | add: print-label; call=print-label-event; caption="Print label";
image="$project-dir$\print-icon-64.png"
```

Add a new user-button labeled "Print label". The button is identified by ID "print-label" and click the button calls the "print-label-event" labeled function in the test-file. By default, the operator is not allowed to click the button (is disabled). The "\$project-dir\$" is a funTEST's internal variable, which will be replaced by the directory of currently active project.

```
#userbtn | set: print-label; enabled=true
```

Enable the "print-label" button - the operator is allowed to click the button.

```
#userbtn | add: manual-release; call=manual-release-event; caption="Release";
enabled=true; image="$project-dir$\release-icon-64.png"
```

Add another user-button labeled "Release" with ID "manual-release".

```
#userbtn | set: *; enabled=false
```

Disable all user-buttons. An asterisk (*) used like an ID selects all buttons to modify.

```
#userbtn | set: print-label; manual-release; enabled=false
```

Disable two specified buttons: "print-label" and "manual-release".

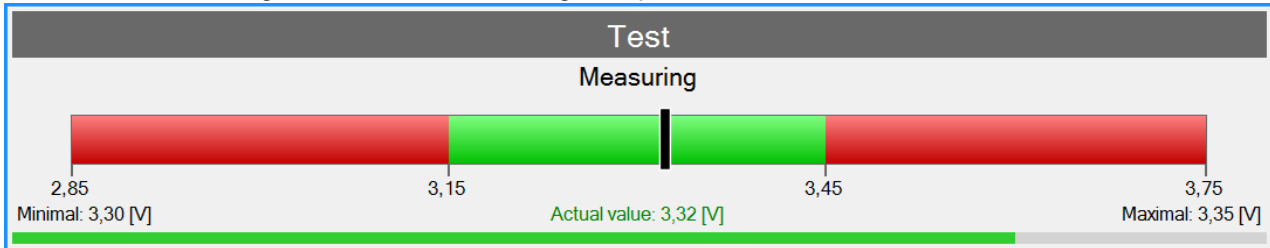
```
#userbtn | remove: *
```

Remove the all user-buttons from the sidebar.

6.4.4.8 #adjust (Value adjust dialog)

```
#adjust | goto| call: <label-name>{: value=[ number] }: title=[ string] {: msg=[ string] }
{: img=[ string] } {: unit=[ string] } {: valuefmt=[ string] }
{: lolim=[ number] } {: hilim=[ number] } {: lorange=[ number] : hirange=[ number] }
{: passcnt=[ int] } {: timeout=[ int] }
```

The #adjust commands enables to automatically repeat the measurement until the value fits between specified limits. While measuring, the value is shown using the operator interface:



There are two ways how to use the #adjust functionality:

- **goto**: repeatedly jump to the measurement section until pass the condition, the measurement value must be linked like an argument
- **call**: repeatedly calls the sub-routine and uses its return value until pass the condition (automatically jumps back to itself)

goto

Typical usage:

Label	Return value	Command	Arguments
measure	3.32	MultimeterDevice	meas: volt?
...
		#adjust	goto: measure: title="Test": value=3.32:...

- sequence starts by a section, which proceed a measurement (plus some calculations, value conversion, and etc.)
- this section must have a label defined
- after this section follows the #adjust command, which is linked to the specified label and measured (or calculated) value
- the #adjust command automatically jumps to measurement section repeatedly, until the measured value is not between specified limits, or timeout occurs
- while the measurement is in progress, the dialog above is displayed and shows the value in real-time
- when the measure value is between limits for a number of following measurements, the #adjust command is done and program continues

call

Typical usage:

Label	Return value	Command	Arguments
...	...	#adjust	call: measure: title="Test":...
...
measure	3.32	MultimeterDevice	meas: volt?
...
		#return	3.32

- the measurement section is outside the main loop (typically after end of test-program), defined like a sub-routine with #return, this routine proceed a measurement and nested calculations
- this section must ends with #return with measured value, which will be passed to calling #adjust

- anywhere the #adjust call be called, funTEST automatically stays on the line with #adjust and repeatedly calls the measurement sub-routine
- while the measurement is in progress, the dialog above is displayed and shows the value in real-time
- when the measure value is between limits for a number of following measurements, the #adjust command is done and program continues

Parameters

label-name	[string]	The target-label to jump, when the measurement has to be repeated.
value	[number]	Linked measured value, can be float with decimal point or integer number. The value is required only when the "goto" way is used.
title	[string]	Dialog title text
msg	[string]	Optional dialog message (below the title) Default: (empty)
img	[string]	Path to picture, optional. Default: (none)
unit	[string]	Value unit, optional. Default: (empty)
valuefmt	[string]	Value format string, optional. Default: 0.00 (two decimal places)
lolim hilim	[number]	Low a high limit to determine the repeating. Only lolim or hilim can be defined (that means - checks value is higher or lower only). At least one of these limits is required.
lorange hirange	[number]	Expected range of value used for bar-graph on the dialog. None of these or both arguments must be passed. If none passed, the range is determined automatically, based on lolim/hilim. Default: (auto)
passcnt	[int]	Number of following measurement, which must be inside limit to close the #adjust function. Default: 5
timeout	[int]	Time to wait for number of following measurements. Default: 0 (infinite time)

Return value

- "0" - successful (measurement pass the condition until timeout occurs)
- "1" - failed (timeout occurs), also the ReturnStatus is set to "1" (error)

Examples

```
#adjust | goto: measure: value=3.32: title="Test": msg="Measuring": uint="V":  
lolim=3.15: hilim=3.45: passcnt=10
```

Start the adjust function with target-label "measure", passing the "3.32" value, value unit and limits.

```
#adjust | none
```

Manually cancel the #adjust operation and hide the dialog.

Parameters

No parameters.

Return value

No return value.

6.4.5 IO

6.4.5.1 #catchio (Wait for a specified IO device input)

```
#catchio | cmd=[ string]; accept=[ string]{; accept2=[ string]; ...; acceptN=[ string]}
      {; timeout=[ number]}{; interval=[ number]}{set-cnt=[ number]}
```

Repetitively send the command to read inputs of IO mapping device and block executing the program until match the expected value.

When #oninput event occurs, the #catchio will break.

Parameters

cmd	[string]	Command of funTEST IO mapping device to read specified inputs. Alias or input numbers can be used, separated by a colon ":".
accept	[string]	Return string sequence of IO command to be accepted. The #catchio command blocks executing the program until one of these string will match the return value. Since funTEST version 1.0.1912.410 the accept argument support wild-cards (? for any one charater and * for any number of charaters).
timeout	[number]	Time limit in [ms] to wait to pass all accept strings. If the timeout is reached, the #catchio command return status will be one and current state of inputs will be returned. Default: (no timeout)
interval	[number]	Interval in [ms] between executing of IO commands. Default: 10 [ms]
set-cnt	[number]	Needed count of internal iteration before "accept filter" will be accepted. Interval between iteration is depended on interval parameter Default: 1

Return value

Last state of inputs in colon-separated format, i.e. "0:1:1".

Returns BREAK when paused while debugging or #oninput event.

Examples

```
#catchio | cmd="r: 0: 2: 4"; accept="1: 1: 1"
```

Read inputs 0, 2 and 4 every 10ms and block program executing until all inputs become logical 1.

```
#catchio | cmd="r: 0: 1"; accept="1: 1": accept="0: 0"
```

Almost the same like before, but more states of inputs are accepted.

```
#catchio | cmd="r: fixture"; accept="0"; interval=200
```

This requires the "fixture" pin alias to be defined. Read the "fixture" input every 200ms and wait until it becomes a logical 0.

```
#catchio | cmd="r: active"; accept="1"; interval=100; timeout=5000
```

This requires the "active" pin alias to be defined. Read the "active" input every 100ms and wait until it becomes a logical 1 for a maximum of 5 seconds. If the "active" input will not change to a log. 1 until timeout is reached, the return status will be set to one and return value to "0" (because of no change).

```
#catchio | cmd="r: in0: in1: break"; accept="1: 0: 0"; accept="?: ?: 1"
```

Wild-card example. This accepts the exact combination of 1:0:0 or any combination with "break" input is active.

```
#catchio | cmd="r:0";accept="1";interval=100;set-cnt=10"
```

Read inputs 0 every 100ms and block program executing until all inputs become logical one - ten times in row.

6.4.5.2 #oninput (External input interrupt)

This command controls external input interrupt. Functionality requires IO mapping to be configured with used inputs enabled. When interrupt is configured, funTEST checks for change of specified input. If there is a transition on input of configured interrupt, the funTEST will go to a specified label or call a specified function.

More than one interrupt can be set. Each interrupt is identified by its input pin number or alias.

Interrupts are bounded to thread from which they were configured. If specific thread finishes, all bounded interrupts to this thread are cleared.

```
#oninput | none
#oninput | disable
```

Disabled configured interrupts of caller's thread. **If this command is executed in the main thread, all interrupts are disabled** (including all another running threads).

```
#oninput | <command>: <label>: pin=[ string/number]: change=[ enum] { : single=[ bool] }
```

Configure (or reconfigure) the interrupt of specified pin number or alias.

Parameters

command	[enum]	What should be done if error or fail has occurred. It can be on of these option: <ul style="list-style-type: none"> • goto - go to line with a specified label, no return • call - asynchronously call specified function block, there is no blocking of running program
label	[string]	Target label to go to or name of function block to call. Function block must ends with #return command.
pin	[string] or [number]	Input pin number or alias that can cause an extern interrupt. It must be defined in teststation's IO mapping.
change	[enum]	Specify a pin level transition. It can be: <ul style="list-style-type: none"> • tohigh - a transition from low to high • tolow - a transition from high to low
single	[bool]	Single-shot option. If true, the configured interrupt is automatically disabled when first occur.

Return value

No return value.

Examples

```
#oninput | goto: start: pin=fixture-closed: change=tohigh
```

This example requires a defined label "start" and input pin with "fixture-closed" alias. If pin value raises from low to high, funTEST will go to on the label "start".

```
#oninput | call: stop: pin=force-stop: change=tolow
```

This example requires a defined function block "stop" and input pin with "force-stop" alias. If pin value falls from high to low, funTEST will call the "stop" function block.

```
#oninput | stop{: <in_0>}...{: <in_N>}
```

Stop interrupt for specified input.

Parameters

`in` [string] or [number] Input pin number or alias to disable the interrupt.

Return value

No return value.

Examples

```
#oninput | stop:force-stop
Disable interrupt for the "force-stop" input.
```

6.4.6 Files

6.4.6.1 #file (File operations)

```
#file | exists?: <filename>
```

Checks if specified file exists.

Parameters

`filename` [string] File-name (full path) to check.

Return value

- "1" if exists
- "0" if not exists

Examples

```
#file | exists?: "c:\\tmp\\file.txt"
Checks if "c:\tmp\file.txt" does exist.
```

```
#file | size?: <filename>
```

Check size of specified file.

Parameters

`filename` [string] File-name to get file-size.

Return value

Size in bytes (number).

Examples

```
#file | size?: "c:\\tmp\\file.txt"
Returns i.e. 1298 (bytes).
```

```
#file | move: <src-filename>: <dst-filename>
```

Move and/or rename the file.

Parameters

src-filename	[string]	Source file-name to be moved.
dst-filename	[string]	Destination file-name.

Return value

No return value.

Examples

```
#file | move:"c:\\tmp\\file.txt":"c:\\file2.txt"
Moves the file "file.txt" from the c:\tmp directory to root of the c:\ and rename it to "file2.txt"
```

```
#file | empty:<filename>
```

Clear all the content of specified file. If the destination file does not exist, the empty file is created.

Parameters

filename	[string]	Destination file to be cleared.
----------	----------	---------------------------------

Return value

No return value.

Examples

```
#file | empty:"c:\\tmp\\file.txt"
Clear or create an empty file "c:\tmp\file.txt"
```

```
#file | del:<filename>
```

Delete the file. If the file does not exists, nothing happen.

Parameters

filename	[string]	Destination file to be deleted.
----------	----------	---------------------------------

Return value

No return value.

Examples

```
#file | del:"c:\\tmp\\file.txt"
Delete file "c:\tmp\file.txt".
```

6.4.6.1.1 INI files

```
#file | ini:read:val=[ string]{;sec=[ string]};file=[ string];;<default>}
#file | ini:write:val=[ string]{;sec=[ string]};file=[ string];;<value>
```

Simplified INI-file access by section and value names.

Parameters

val	[string]	Value name of specified section to be read or written.
sec	[string]	Optional section name. If no section is passed, the values are placed to the default section to start of the INI file without the

		"[SECTION]" specification.
file	[string]	Optional file name of INI file. If no frame passed, the default INI file is used - same location and name like the test-file, accessing the INI file.
default	[string]	The default value to be return when the INI-file, specified section or specified value name does not exist. If no default value is specified and the value does not exist, the method will return by Return status = 1.
value	[string]	The value to be written to specified section/value name.

Return value

When reading from INI by `ini:read`, the value of specified section/value or default value is returned.

Examples

```
#file | ini:read:val=Serial:sec=MAIN:"1"
Read value "Serial" from section "MAIN". When does not exist, return "1".
```

```
#file | ini:write:val=Serial:sec=MAIN:file="c:\\settings.ini":"2"
Write "2" string to value "Serial" in "MAIN" section. The file specified section/value are created if they do not exist.
```

6.4.6.1.2 Text files

```
#file | text:lines?:file=[ string]
```

Count lines in the source file.

Parameters

file	[string]	Path to source text-file.
------	----------	---------------------------

Return value

Number of lines (integer).

Examples

```
#file | text:lines?:file="c:\\tmp\\file.txt"
Count and return lines of "c:\tmp\file.txt" file.
```

```
#file | text:length?:file=[ string]
```

Number of characters in the source file.

This is different to [size?](#) command, because the size is byte-oriented and `text:length?` is character-oriented.

Parameters

file	[string]	Path to source text-file.
------	----------	---------------------------

Return value

Number of characters (integer).

Examples

```
#file | text:length?:file="c:\\tmp\\file.txt"
Return number of characters of "c:\tmp\file.txt" file.
```



```
#file | text:read?:file=[ string]
```

Read all content of specified text-file.

Parameters

file	[string]	Path to source text-file.
------	----------	---------------------------

Return value

Content of source file.

Examples

```
#file | text:read?:file="c:\\tmp\\file.txt"
Return content of "c:\tmp\file.txt" file.
```

```
#file | text:readln?:file=[ string]{:ln=[ int]}{:expr=[ string]}{:expr-last=[ string]}
```

Read specified line of source text file.

Parameters

file	[string]	Path to source text-file.
ln	[int]	Index of line to be read (zero-based). If negative, line at the end is taken (where -1 is the last line). Default: 0 (first line)
expr	[string]	Return first (expr) or last (expr-last) line from text-file matching the regular expression. Default: none (by ln arg)
expr-last		

Return value

Specified line of source file. If line does not exists, an empty string is returned.

Examples

```
#file | text:readln?:file="c:\\tmp\\file.txt":ln=2
Return 3rd line from the beginning of c:\tmp\file.txt
```

```
#file | text:readln?:file="c:\\tmp\\file.txt":ln=-1
Return last line of c:\tmp\file.txt
```

```
#file | text:readln?:file="c:\\tmp\\file.txt":ln=-2
Return 2rd line from the end of c:\tmp\file.txt
```

```
#file | text:readln?:file="c:\\tmp\\file.txt":expr="index[0-9]+"
Return first line which contains "index" and any number (e.g. index0, index29, index 568.. etc)
```

```
#file | text:write:<text>:file=[ string]
#file | text:append:<text>:file=[ string]
#file | text:insert:<text>:file=[ string]{:offset=[ int]}
```

Write/append/insert the text to destination file. If writing, the content of the file is **overwritten**.

Parameters

text	[string]	Content to write/append/insert to the destination file.
file	[string]	Path to destination text-file.

`offset` [int] Character offset in the file to insert the text. Zero-based.
Default: 0 (beginning of the file)

Return value

No return value.

Examples

```
#file | text:write:"hello world!":file="c:\\tmp\\file.txt"
```

Write the string "hello world!" (12 characters) to file c:\tmp\file.txt, overwriting its previous content.

```
#file | text:append:"end of the file":file="c:\\tmp\\file.txt"
```

Append the string "end of the file" (15 characters) at the end of the file c:\tmp\file.txt.

```
#file | text:insert:"new ":file="c:\\tmp\\file.txt":offset=6
```

Insert the string "new " (4 characters) to the file c:\tmp\file.txt of the first example, the new content of the file will be "hello new world!".

```
#file | text:writeln{<line_0>:...<line_N>}:file=[ string]
#file | text:appendln{<line_0>:...<line_N>}:file=[ string]
#file | text:insertln{<line_0>:...<line_N>}:file=[ string]{:offset=[ int]}
```

Write/append/insert lines (strings, separated by a system standard new-line delimiter) to destination file. If writing, the content of the file is **overwritten**.

Parameters

<code>line₀ to line_N</code>	[string]	Line(s) to write/append/insert to the destination file.
<code>file</code>	[string]	Path to destination text-file.
<code>offset</code>	[int]	Line offset in the file to insert lines. Zero-based.

Return value

No return value.

Examples

```
#file | text:writeln:"funTEST":"system":file="c:\\tmp\\file.txt"
```

Write 3 lines "funTEST" and "system" to file c:\tmp\file.txt, overwriting its previous content.

```
#file | text:appendln:"solution":file="c:\\tmp\\file.txt"
```

Append 1 line "solution", to file c:\tmp\file.txt.

```
#file | text:insertln:"testing":file="c:\\tmp\\file.txt":offset=1
```

Insert 1 line "solution" after the first line to file c:\tmp\file.txt of the first example, the new content of the file will be 3 lines "funTEST", "testing" and "system".

```
#file | text:remove:from=[ int]{:to=[ int]}{:len=[ int]}:file=[ string]
#file | text:removel:from=[ int]{:to=[ int]}{:len=[ int]}:file=[ string]
```

Remove a number of characters/lines from the destination file.

Parameters

<code>from</code>	[int]	Start of range of characters/lines to be removed. Zero-based.
<code>to</code>	[int]	End of range of characters/lines to be removed. Zero-based.
		Default: see note below

len	[int]	Number of characters/lines to be removed. Default: see note below
file	[string]	Path to destination text-file.

Note: if none of 'to' and 'len' arguments are not passed, the len=1 is supplied (1 character/line to be removed). Otherwise - pass one of the 'to' or 'len' argument to perform required action.

Return value

No return value.

Examples

```
#file | text:remove:file="c:\\tmp\\file.txt"
Remove first character in the file c:\tmp\file.txt
```

```
#file | text:removel:file="c:\\tmp\\file.txt"
Remove first line in the file c:\tmp\file.txt
```

```
#file | text:remove:from=1:to=3:file="c:\\tmp\\file.txt"
Remove characters in the range 1 to 3 (3 characters in total) in the file c:\tmp\file.txt
```

```
#file | text:removel:from=1:len=2:file="c:\\tmp\\file.txt"
Remove total of 2 lines, starting at index 1 (second line) in the file c:\tmp\file.txt
```

6.4.6.2 #export (Export to file)

Allows to export a range of specified sheet to following formats:

- CSV
- PDF

6.4.6.2.1 CSV

```
#export | csv{:sheet=[ string]}{:from=[ string]}{:to=[ string]}:path=[ string]
{:delimiter=[ string]}{:rowfilter=[ bool]}{:append=[ bool]}
```

Export cell values of range of specified sheet to a CSV file (text file with separated values). Target file will be overwritten by default. Target directory will be automatically created (including sub-folders) if does not exist.

Parameters

sheet	[string]	Source sheet Default: TEST
from	[string]	Start of exported range. Cell name has to be used - i.e. "A1". Default: A1
to	[string]	End of the exported range. Same format like "from" is used.
path	[string]	Path of the destination file to export.
delimiter	[string]	Value delimiter of CSV file. Default: ";" (semi-colon)
rowfilter	[bool]	If true, the first column of the (from .. to) source cell range is used like a row filter. If the column's value is "0", the row is not exported. Any other value (including empty cell) will cause the row to be exported. The value of the first column is not included in the result.
append	[bool]	When true, the target file is not overwritten, but the content is appended. When the file does not exist, it will be created. Default: false

Return value

No return value.

Examples

```
#export | csv:sheet="file":to=C3:path="$project-dir$\\example1.csv"
Export values from sheet "file", range A1 to C3 to the file "example1.csv", located in project's directory.
```

```
#export | csv:from=B2:to=D4:path="c:\\example2.csv":delimiter=","
Export values from sheet "TEST", range B2 to D4 to the file "c:\\example2.csv" (fixed path). A comma will be used to separate values on lines in the csv file.
```

6.4.6.2.2 PDF

```
#export | pdf:sheet=[string]:path=[string]{:from=[string]:to=[string]}
{:range=[string]}
```

Export cells (including formatting) of range of specified sheet to a PDF file. Target file will be overwritten by default. Target directory will be automatically created (including sub-folders) if does not exist.

Notes: the output PDF file respects the configured page formatting (e.g. header, footer, page numbering, margins, ..) - this must be done directly in the Calc

Parameters

sheet	[string]	Source sheet
path	[string]	Path of the destination file to export.
from	[string]	Start of exported range. Cell name has to be used - e.g. "A1". Default: A1
to	[string]	End of the exported range. Same format like "from" is used.
range	[string]	Full range specification in format "<from>:<to>", e.g. "A1:D5" Default: from and to arguments

Return value

No return value.

Examples

```
#export | pdf:sheet="LABEL":from=A1:to=C3:path="$project-dir$\\label1.pdf"
Export PDF from sheet "LABEL", range A1 to C3 to the file "example1.pdf", located in project's directory.
```

```
#export | pdf:sheet="LABEL":range="A1:D4":path="c:\\example2.pdf"
Export PDF from sheet "LABEL", range A1 to D4 (range argument used) to the file "c:\\example2.pdf" (fixed path).
```

6.4.6.3 #stat (Statistics)

Assisted making of statistics.

6.4.6.3.1 CSV

The statistics is based on internal key-value buffer, which is filled-up by single commands and then append to a specified file. This buffer is automatically cleared when starting the test-file.

```
#stat | csv:init:<name0>=<value0>: ...: <nameN>=<valueN>
```

Init persistent values, which will not be deleted by "clr" command. They will be listed first to the output file.

Parameters

name	[string]	Name of the key
value	[string]	Value of the key

Return value

No return value.

Examples

```
#stat | csv:init:Operator="$user-name$":Station="$teststation-name$"
Set "Operator" and "Station" persistent values. In this example, the standard variables are used. Any text can be used here.
```

```
#stat | csv:c
```

Clears all values from the buffer. The init values stay untouched.

Parameters

No parameters.

Return value

No return value.

```
#stat | csv:v:<name>:<value>
```

Add a new key or modify a value of an existing key. If the specified key does not exist, it will be inserted at the end of the list. When key exists, its value is modified.

Parameters

name	[string]	Name of the key
value	[string]	Value of the key

Return value

No return value.

Examples

```
#stat | csv:v:"Resistor R1": "965.5"
Setup new or modify "Resistor R1" key.
```

```
#stat | csv:save:<path>
```

Append the current record to the destination file. If the file does not exist, it will be created with header, specified by keys.

Parameters

path	[string]	Path to destination file
------	----------	--------------------------

Return value

No return value.

6.4.6.3.2 AMSTAT

Control of automated statistics.

```
#stat | amstat{:enabled=[ bool]}
```

Set parameters.

Parameters

enabled	[bool]	Override the "Enabled" definition in the AMSTAT sheet. Default: (current value - no change)
---------	--------	---

Return value

No return value.

Examples

```
#stat | amstat:enabled=no
Disable the AMSTAT by command.
```

6.4.6.4 #statrec (Basic statistics record)

```
#statrec | <stat-file-path>{;src=[ int] }{;sheet=[ string]}
```

Create a new statistic file if not exist and add a new record. The file **is not** saved on each added record automatically. To save the file, use the [#statsave](#) command.

Parameters

stat-file-path	[string]	Path to the statistic-records file. If no path is passed, default daily-statistic will be automatically created.
src	[int]	The source line of statistics sheet. Default: 1
sheet	[string]	The source sheet of statistics. Default: STAT

Return value

No return value.

Examples

```
#statrec
Create a default (daily) statistic file in the project's "statfiles" sub-directory. The name will be "Dyyyy-MM-dd<test-file>.csv", where "yyyy" is the year, "MM" month, "dd" day and "test-file" is the currently opened test-file. Every day a new-one will be created.
```

```
#statrec | "c:\\temp\\stat.csv"
Create a new statistic file "c:\\temp\\stat.csv" if not exist and add a new record. Only this file will be used.
```

6.4.6.5 #statsave (Basic statistic save)

```
#statsave
```

Saves currently used statistic file to a hard-drive.

Parameters

No parameters.

Return value

No return value.

6.4.7 Test-file

6.4.7.1 #cellread, #cellwrite, #cellerase, #cellcopy (Cell direct-access)

```
#cellread/#cr | { sheet=[ string]: } <cell>
#cellwrite/#cw | { sheet=[ string]: } <cell>{ : <value1> } { : <value2> } ... { : <valueN> }
#cellerase/#ce | { sheet=[ string]: } from=[ string]: to=[ string]
```

Provide a direct read or write access to specified cell of the test-program.

Parameters

sheet	[string]	Source/target sheet name. Default: TEST
cell, from, to	[string]	Source/target cell specification. Cell can be typed directly or like an expression using column names, labels and offsets. <i>Resulting column must be always in range "A" to "AMJ" (1 to 1024 column) and row must be in range 1 to 65536.</i> Possible cell formats: <ul style="list-style-type: none"> • Direct <column-letter(s)><row-number> - "column-letter(s)" is from "A" (first column) to "AMJ" (last 1024th column) - "row-number" is from 1 to 65536. <i>Example:</i> B5 - second column, fifth row • Direct column with offset (\$<column-letter(s)> +/-<offset>) <row-number> - "column-letter(s)" and "row-number" are same like above - "offset" is an integer number The dollar-sign ("\$") must be present here and together with column letter(s) and offset must be surrounded by parentheses. <i>Examples:</i> (\$A+2) 4 = "C4" (\$E-1) 6 = "D6" (\$B-2) 1 => error, because first column is "A" and this expression points before the "A" column • Using column names and labels (<column-name> { +/-<offset> }) (<label-name> { +/-<offset> }) - "column-name" is the name of column, column-names are defined by the first row in the sheet - "label-name" is the target label, specified by a column named "Label" in the test-program, label-name can be also a key-word "@this", which means the currently processed row by the funTEST. <i>Label names can be used only on the TEST sheet.</i> - "offset" is an optional parameter with the same functionality like above

		<p>Expression can be combined. It means, you can use column name and direct row, or direct column and label-name like a row, etc...</p> <p><i>Examples:</i> (following examples expecting a "test" label on line 10, "mycol" on column "H" and current line 20) A(test+1) = direct column, using label "test" with offset +1 => A11 (mycol-1) 2 = using column name with offset -1 and direct row => G2 (mycol) (test) = using column name and also label, no offsets => H10 (mycol) (@this+1) = using column name and current line with offset +1 => H21 X(@this) = using direct column and current line, no offsets => X20 X(@this) on the line below => X21, etc... (\$K+1) (test) = using direct column with offset +1 and label => L10</p>
value _{1..N}	[string]	Value which will be written into the target cell. If no value is passed, the target cell is cleared. If more than one value is passed, values are first merged together and the result will be written into the target cell.

Return value

#cellread - returns a value in the source cell
 #cellwrite, #cellerase - no return value

Examples

```
#cellread | X1
```

Simple read a value from the cell "X1".

```
#cellread | sheet=example; A2
```

Read a value from the cell "A2" on the sheet "example".

```
#cellread | (mycol+2)(@this)
```

Read a value from the cell, specified by column-name "mycol" with offset +2 and currently processed row.

```
#cellwrite | X1
```

Clear the cell "X1".

```
#cellwrite | X1;"myvalue"
```

Direct write a value "myvalue" to the cell "X1".

```
#cellwrite | sheet=example;(mycol)2;"ab";"cd";"ef";"gh"
```

First, the parameters after cell specification are merged => "abcdefgh". This value will be written into the column named "mycol", second row.

```
#cellerase | sheet=example;from=A1;to=B5
```

Clears the cell-range from A1 to B5 on sheet "example".

```
#cellcopy/#cc | {srcsheet=[ string]}{:dstsheet=[ string]}:srcfrom=[ string]
:srceto=[ string]:dst=[ string]{:convert=[ enum]}
```

Copy the area of cells from source to destination, optionally with automatic value conversion.

Parameters

<code>srcsheet, dstsheet</code>	[string]	Source/target sheet name. Default: TEST
<code>srcfrom, srcto, dst</code>	[string]	Source area and destination starting-cell specification. The format is the same like the #cellread/write/erase commands.
<code>convert</code>	[enum]	Automatic cell conversion while copying. If the source value cannot be converted, the empty string is passed. <ul style="list-style-type: none"> • <code>int</code> - convert to integer • <code>double</code> - convert to floating point Default: (no conversion)

Return value

No return value

Examples

```
#cellread | dstsheet=VALUES:srcfrom=W2:srcto=X10:dst=A1:convert=double
```

Copy the range from W2:X10 from the TEST sheet (no srcsheet argument) to VALUES destination sheet starting the cell A1. Automatic conversion to double will be performed.

6.4.7.2 #retclear (Clear return values)

```
#retclear | {from=[string]}{;to=[string]}
```

Clear a specified range of Return Value and Return Status column in the test-program spreadsheet. This should be usually done at the start of the test-program.

Parameters

<code>from</code>	[string]	An expression to specify starting line of range to be cleared. If omitted, the first row of test-program is used. Possible options: <ul style="list-style-type: none"> • line number greater than 1 • expression using variable @this or label name and offset using + or - sign
<code>to</code>	[string]	An expression to specify ending line of range to be cleared. If omitted, the last row of test-program is used.

Return value

No return value.

Examples

```
#retclear
```

Clear all return values in the whole test-program.

```
#retclear | from=@this
```

Clear return values in the current line (means line with #retclear command) to the end of the test-program.

```
#retclear | from=10
```

Clear return values in the fixed line number 10 to the end of the test-program.

```
#retclear | from=@this+1;to=@this+10
```

Clear return values in the next 10 lines of current row.

```
#retclear | from=labelA+5
```

Clear return values in 5 lines from line labeled "labelA" to the end of the test-program.

```
#retclear | from=labelA; to=labelB
```

Clear return values in line labeled "labelA" to the line labeled "labelB".

6.4.7.3 #testfile (Testfile control)

```
#testfile | close
```

Closes the currently running test-program. At this point the **execution of test-file is broken** and no following instruction is executed.

Parameters

No parameters.

Return value

No return value.

```
#testfile | selectnew
```

Shows up the test-file dialog to select a new test-file. The previous project is used. If no new test-file is selected, the "Return Status" = 1 is set and following instructions are executed normally. If a new test-file is properly selected, the execution of current test-file is broken.

Parameters

No parameters.

Return value

No return value.

```
#testfile | save
```

Immediately saves the currently running test-file to a hard-drive.

Parameters

No parameters.

Return value

No return value.

```
#testfile | reload
```

Close and re-open the project, using current test-file. At this point the **execution of test-file is broken** and no following instruction is executed.

Parameters

No parameters.

Return value

No return value.

```
#testfile | load{:<project>}:<test-file>{:force=[bool]}
```

Close current, load and start new test-file by specified name of test-file and optionally project. At this point the **execution of test-file is broken** and no following instruction is executed.

Parameters

project	[string]	Project name or full path to .ftproject.xml project's file. If no project is specified, the test-file is about to search in all projects.
test-file	[string]	Default: none Test-file name or full path to test-file (.ods). If no project is specified and there is more than one test-file with the same name, and error is thrown. By default, if project to be loaded is already loaded, the command will abort.
force	[bool]	Force project to be reloaded, even if it is the same like currently loaded.

Default: no

Note: path or name of project and test-file is case non-sensitive

Return value

"LD" if the project is already loaded, otherwise No return value

Examples

```
#testfile | load:tf01
```

Load program with "tf01" name in all projects.

```
#testfile | load:prj01:tf01
```

Load program with "tf01" name in "prj01" project.

```
#testfile | load:"c:\projects\prj01\tf01.ods"
```

Load program by full-path of test-file. The test-file must be assigned in any existing projects.

```
#testfile | load:"c:\projects\prj01\prj01.ftproject.xml":"c:\projects\prj01\tf01.ods"
```

Load program by full-path of project and test-file.

6.4.7.4 #str (String operations)

Request funTEST version: 1.0.1906.311

6.4.7.4.1 Split

```
#str | split{:<src>}{:dev=[string]:<cmd>}{:var=[string]}{:sheet=[string]}:cell=[string]
{:sep0=[string]:...:sepN=[string]}{:rsep0=[string]:...:rsepN=[string]}
{:convert=[enum]}
```

Split string by separator(s) to spreadsheet columns/rows. Destination cells are always overwritten.

Parameters

src	[string]	Source string to split. If no string is passed, the function
-----	----------	--

		automatically take the ReturnValue from previous valid step. Default: ReturnValue from previous step
dev, cmd	[string]	Get source string directly from any device using a command. If this argument is not passed, the function try take the value directly from src argument. <ul style="list-style-type: none"> • dev - alias of device (must be defined in active teststation) • cmd - command to send to a device Default: src argument
var	[string]	Get source string directly from any testfile variable. The source variable can also be an element of an array, use the [i] index suffix to specify. If this argument is not passed, the function try to take the value directly from src argument. Default: src argument
sheet	[string]	Target sheet to store splitted values. Default: TEST
cell	[string]	Target top-left cell to store values. Absolute or relative format possible, same format like #cellread/#cellwrite/#cellerase functions.
sep, rsep	[string]	Column (sep) and row (rsep) separator. One of them or both arguments can be used at the same time to split source string to columns or rows or both. There can be multiple (alternative) column or row separators.
limit, rlimit	[int]	Column and row limits. These arguments will limit a max number of columns and/or rows. Default: no limit
convert	[enum]	Automatic conversion of all values. If the conversion is not possible, the source value is passed to the cell. <ul style="list-style-type: none"> • int - convert to integer numbers • double - convert to double numbers Default: (no conversion)

Return value

No return value. Results are directly written to another cells.

Examples

```
#str | split:src="text;to;parse\nnew;row":cell="W(@this)":sep=";":rsep="\n"
```

Split directly passed source string to "W"-column at processed row using ";" column separator and "/" row separator.

Result:

	W	X	Y
100	text	to	parse
101	new	row	

(row numbers are for example, depends on current processing step)

```
#str | split:sheet=VALUES:cell=A1:sep=";":rsep="\n":rsep="\r"
```

Split ReturnValue of previous valid step to "VALUES" sheet, starting by A1 cell. Multiple column/row separators are used.

```
#str | split:sheet=VALUES:cell=A1:sep=";":convert=int
```

Split ReturnValue of previous valid step to "VALUES" sheet, starting by A1 cell. Automatic conversion of values to integer will be done.

6.4.7.4.2 At

```
#str | at{<src>}{:var=[ string] }:sep=[ string]
      {:rsep=[ string] }{:cell=[ int] }{:row=[ int] }
```

Internally splits the string into the virtual table and returns a value by row and column index.

Parameters

<code>src</code>	[string]	Source string to split. If no string is passed, the function automatically take the ReturnValue from previous valid step. Default: ReturnValue from previous step
<code>var</code>	[string]	Get source string directly from any testfile variable. The source variable can also be an element of an array, use the [i] index suffix to specify. If this argument is not passed, the function try to take the value directly from <code>src</code> argument. Default: <code>src</code> argument
<code>sep, rsep</code>	[string]	Column (<code>sep</code>) and row (<code>rsep</code>) separator. Column separator is needed, but row separator is optional. One of them or both arguments can be used at the same time to split source string to columns or rows or both. There can be multiple (alternative) column or row separators. Default: <code>rsep</code> ignored
<code>cell, row</code>	[int]	Cell/Row absolute index from 0 to return. Default: 0

Return value

String value indexed by arguments row and cell.
Error is returned, when the row or cell index is out of range.

Examples

```
#str | at:src="text; to; parse\nnew; row": sep="; ": rsep="\n": row=0: cell=0
Returns the value "text".
```

```
#str | at:src="text; to; parse\nnew; row": sep="; ": rsep="\n": row=0: cell=2
Returns the value "parse".
```

```
#str | at:src="text; to; parse\nnew; row": sep="; ": rsep="\n": row=1: cell=0
Returns the value "new".
```

6.4.7.5 Localization

Since version 1.0.1912.1913 the funTEST supports test-file localization. The localization is stored in external .local.xml file, which can be loaded using dedicated command.

The localization file is a standard FPC XML localization format (v2.0). There is a dedicated software to create and edit localization files: FPC Localization Editor

Following is supported:

- Step Name automatic translation
- Text of [#dlg](#) and [#msg](#) automatic translation
- Variable caption automatic translation
- Manual translation using [#local](#) command

Be careful while choosing the localization key prefix and suffix in the Localization Editor. It's recommended to use "#" both for prefix and suffix. For example it's not recommended to use "\$", because this letter is used for

inline [variables](#) names and also "@", because it's used for some keywords. Of course if you use it even this recommendation, it will work, but you have to double the localization prefix or suffix character to type it (e.g. @@ will be translated to a single @ after translation and so on).

6.4.7.5.1 #local (Localization)

```
#local | load: <path>
```

Loads the xml localization file and selected the default localization according the funTEST's language. Variables are also reloaded to refresh their captions in the operator's interface.

Parameters

path	[string]	Path to .local.xml localization file. It's possible to use <code>def</code> keyword to load default file - the same name test-file, but with .local.xml file extension.
------	----------	--

Return value

No return value.

Examples

```
#local | load: "c:\\files\Test.local.xml"
```

Load "Test.local.xml" file from "c:\files" directory.

```
#local | load: def
```

Let's have a test-file name "Board1.ods" and project "Boards".
This command will load the "c:\Users\Public\FPC\funTEST\projects\Boards\testfiles\Board1.local.xml".

```
#local | lang: <name>
```

Select another language like default.

Parameters

name	[string]	The short 3-letter name of language, e.g. "cze", "eng", ...
------	----------	---

Return value

No return value.

```
#local | {<lang>:} <text>
```

Localize the text using default or explicitly selected language. If no localization file is loaded, the funTEST will automatically try to load the default localization file (test-file name with .local.xml extension).

Parameters

lang	[string]	Use defined language to translate. Default: funTEST's language
text	[string]	Any text to translate.

Return value

Translated text

Examples

Translation expect the file to be defined. Translated text in following examples are for illustration only.

```
#local | "@CompanyName@: FPC s. r. o. "
```

Translate text using default language (english in this case) to "Company name: FPC s. r. o. ".

```
#local | cze: "@CompanyName@: FPC s. r. o. "
```

Translate text to czech: "Jméno společnost: FPC s. r. o. ".

6.4.8 Printing

```
#print | *def{:printer=[string]}{:?<var0>=[string]:...:?<varN>=[string]}
```

Set printing defaults:

- default printer name
- create/overwrite variables

Variable(s) are replaced in the final printer's data. To define variable in the source data, use a variable name between dollar-signs: `$variable$`

Parameters

<code>printer</code>	[string]	Printer name to be used as default (e.g. "ZDesigner GX430t", the name in the Windows Printers control panel).
<code>var_N</code>	[string]	Variable(s) to be created/changed. Each variable name must start by '?', otherwise it is not recognized as variable. If variable does not exist, its created, otherwise the value of existing one is changed.

Return value

No return value.

Examples

```
#print | *def:printer="ZDesigner GX430t":?header="Hello! ":?footer="Good bye.. "
Set default printer and create/change two variables "header" and "footer".
```

6.4.8.1 Labels

Send an unformatted data to the specified printer (RAW printing). This allows typically to control label-printers by sending printer's command directly.

Source data can be directly in test-file or in an external text-file. This depends on which of "sheet" or "file" argument of "label" command is used:

- `sheet` used: `range` argument is required, data are in the specified sheet and range
- `file` used: data are in specified external text-file

[Variables](#) (standard and test-file) are applied to final data before sending to the printer.

```
#print | label:sheet=[string]:range=[string]{:printer=[string]}
```

Print using data defined directly in the test-file's sheet.

Parameters

<code>sheet</code>	[string]	Sheet name with print data.
<code>range</code>	[string]	Range of specified sheet to obtain source data. Format: "<cell-from>: <cell-to>" (i.e. "A1: B10")

		<ul style="list-style-type: none"> • one column only: all rows are merged to final printer's data (using "\n" separator, skipping empty rows) • more than one column: <ul style="list-style-type: none"> - first column is used like a row-filter: "1" means data-row is always taken, empty means data-row is taken if non-empty, otherwise ignored - following columns are data: merged without any separator from all cells of the row
printer	[string]	Optionally set a printer name. Required if no default printer set.

Default: printer set by [*def](#)

Return value

No return value.

Examples

Following examples uses print commands of Zebra label printers. Sheet: "PRINT" for example

	A
1	^XA
2	^PW1020
3	^ADN,80,40
4	^FO10,30
5	^FDfunTEST^FS
6	^XZ
7	
8	
9	
10	

```
#print | label:sheet="PRINT":range="A1:A10"
```

Print a simple one-text only label using printer's data from sheet "PRINT", range A1 to A10 (one column, 10 cells in total) to the printer. Rows 7 to 10 are ignored, because they are empty. Rows 1 to 6 are merged to one string using LF ("\n") separator.

	A	B
1		^XA
2		^PW1020
3	1	^ADN,80,40
4	1	^FO10,30
5	1	^FDfunTEST^FS
6	0	^ADN,50,30
7	0	^FO10,140
8	0	^FDTesting system^FS
9		^XZ
10		

```
#print | label:sheet="PRINT":range="A1:B10"
```

Print a simple label with a header and optionally one text below, using printer's data from sheet "PRINT", range A1 to B10 (column A is used like a row-filter, column B like data) to the printer. Rows 1, 2 and 9 are always taken (because they are not empty and there is no filtering specified), rows 3 to 8 are optionally taken/ignored, depending on column "A" of corresponding row. Row 10 is always ignored, because there is no filtering specified and data are empty.

```
#print | label:file=[ string]{:printer=[ string]}
{: ?<var0>=[ string]:?...: ?<varN>=[ string]}
```


Print using external text file with variable replacement.

Parameters

file	[string]	Path to text-file with print data.
printer	[string]	Optionally set a printer name. Required if no default printer set. Default: printer set by *def
var _N	[string]	Optional additional variables to be replaced in the loaded data from the file. Variable names must start by "?" to recognize argument(s) as variables.

Return value

No return value.

Examples

Expected example file content:

```
^XA
^PW1020
^ADN, 80, 40
^FO10, 30
^FDfunTEST^FS
^ADN, 50, 30
^FO10, 140
^FDCode: $code$^FS
^ADN, 50, 30
^FO10, 230
^FDOperator: $user-login$^FS
^XZ
```

```
#print | label:file="$project-dir$\\labels\\$testfile-name$.txt":?code=123456789
Print label, defined in the .txt file with same name like current test-file in the "\labels" sub-directory of
currently loaded project. Additional variable $code$ is replaced and $user-login$ is replaced from standard
variables.
```

6.4.8.2 Text

Send a plain text to the specified printer to print.

Usage is the same like [Label printing](#), see for additional arguments only.

```
#print | text:sheet=[ string]:range=[ string]{:printer=[ string]}
{:page-margin=[ int]}
```

Print using data defined directly in the test-file's sheet.

Additional parameters

page-margin	[int]	Page margins in [mm]. Default: 10 [mm]
-------------	-------	--

```
#print | text:file=[ string]{:printer=[ string]}{:page-margin=[ int]}
{:?<var0>=[ string]:...:?<varN>=[ string]}
```

Print using external text file with variable replacement.

Additional parameters

The same like above.

6.4.8.3 Spreadsheet

```
#print | sheet: <sheet>{: printer=[ string] }{: format=[ enum] }{: orientation=[ enum] }
```

Directly print the selected OpenOffice Calc's sheet. This is the same like "Print" functionality in the OpenOffice Calc.

Requirements:

- It is necessary to define "Print sections" (the Format menu) in the test-file (.ods) - you have to remove them all on all sheets, or define a new on specified sheet to be printed otherwise nothing will happen after printing (no pages will be send to the printer)

Parameters

sheet	[string]	Spreadsheet name to print.
printer	[string]	Optionally set a printer name. Required if no default printer set. When a network printer is used - the name must be an UNC path, e.g. "\\printserver\HP LaserJet 1000" (backslashes must be escaped). Take care: if specified printer does not exists or the name is wrong, the OpenOffice will print the document on the default printer. Default: printer set by *def
format	[enum]	Paper format: <ul style="list-style-type: none"> • A3 • A4 • A5 • B4 • B5 Default: A4
orientation	[enum]	Paper orientation: <ul style="list-style-type: none"> • portrait • landscape Default: portrait

Return value

No return value.

Examples

```
#print | sheet: "PRINT": printer="PDFCreator"
```

Print the sheet "PRINT" on "PDFCreator" printer using standard A4 format - portrait.

```
#print | sheet: "PRINT": printer="\\\\printserver\\HP LaserJet 1000":  
format=A5:orientation=landscape
```

Print the sheet "PRINT" on a network printer using A5 paper format - landscape.

6.4.9 Special

6.4.9.1 #catch (Wait for a specified response of device)

```
#catch | dev=[ string]:cmd=[ string]:accept=[ string]{:accept2=[ string]:...:acceptN=[ string]}
      {;timeout=[ number]}{;interval=[ number]}{;inv=[ bool]}
```

Repetitively send the command to specified device and block executing the program until return value match to expected value.

When #oninput event occurs, the #catch will break.

If any command execution to the device fails, the #catch loop breaks and error message of failed command is passed to Return Value and Return Status set to 1.

Parameters

dev	[string]	Target device alias.
cmd	[string]	Command to send to target device.
accept	[string]	Return value(s) of command to be accepted. The #catch command blocks executing the program until any of these strings match the return value. The accept argument support wild-cards (? for any one charater and * for any number of charaters).
timeout	[number]	Time limit in [ms] to wait to pass all accept strings. If the timeout is reached, the #catch command return status will be one and current state of inputs will be returned. Default: (no timeout)
interval	[number]	Interval in [ms] between executing of IO commands. Default: 100 [ms]
inv	[bool]	Inverts condition defined by parameter accept.

Return value

The matching return value of device.

Returns `BREAK` when paused while debugging or #oninput event.

Examples

```
#catch | dev="tester":cmd="finish?";accept="1"
```

Send the "finish?" command to the device with alias "tester" until the command returns "1". There is no timeout.

```
#catch | dev="tester":cmd="finish?";accept="1":timeout=5000
```

Almost the same like previous, but with defined timeout of 5 seconds. If the command will not return "1" within 5 seconds the #catch command is terminated with error - Return Status will be set to 1.

```
#catch | dev="com":cmd="read?";accept="*PASS*"
```

Wild-card example, the "read?" command is send to "com" device as long until the return value will contain "PASS" sub-string.

6.4.9.2 #login (Login operations)

```
#login | login{:user=[ string]}{:pwd=[ string]}{:title=[ string]}
```

Perform the login or raise the login dialog. This will affect the global logged user in funTEST.

Parameters

user	[string]	Username to login. Can be omitted if specified plugin is configured to login via password only.
------	----------	---

<code>pwd</code>	[string]	Default: empty Password of user to login.
<code>title</code>	[string]	Default: empty Custom title of login dialog Default: "Login" (translated according localization)

Return value

1 or 0 - user logged in successfully or not (dialog canceled)

Examples

```
#login | login
Just raise the login dialog.
```

```
#login | login: user="admin": pwd="adm"
Direct user login via username and password. No dialog is raised.
```

```
#login | try{:title=[string]}{:checkright=[string]}
```

Raise the login dialog and returns the logged-in user. This will NOT affect the global logged user.

Parameters

<code>title</code>	[string]	Custom title of login dialog Default: "Login" (translated according localization)
<code>checkright</code>	[string]	Check for specified right of logged-in user. Default: -

Return value

If the login dialog is cancelled:

- CANCEL

If there is "checkright" argument:

- 1 or 0 (logged operator has the required right or not)

Otherwise:

- <login-name>: <username>: <rights>

"rights" are passed separated by a comma, keywords are `oper`, `prg` and `adm`

Examples

```
#login | try
Raise the login dialog with default title and return logged-in login-name, username and rights.
```

```
#login | try:title="NG box confirmation":checkright="adm"
Raise the login dialog with custom title and check if logged user has the right "adm". Returns 0/1.
```

```
#login | plugin:[<subarguments>]
```

Call a user-command in login plugin. The functionality and arguments depends on specified plugin features.

Parameters

<code>subarguments</code>	[string]	Arguments to pass via the user-command.
---------------------------	----------	---

Return value

Depends on specified plugin.

Examples

```
#login | plugin:[ num=1:str="abc"]
```

Call the plugin's user command - pass for example arguments `num` and `str`.

6.4.9.3 #bct (Batch-test)

```
#bct | run: <name>{: panel=[ list] }{: type=[ string] }{: range=[ string] }{: segment=[ string] }
```

Run specified batch-test. The batch-test must be defined in the project. Command blocks the execution until batch-test finishes.

Parameters

<code>name</code>	[string]	Batch-test name to run.
<code>panel</code>	[list]	Filter batch-test definition to only selected panel(s). Enter numbers, comma separated. Default: (all panels)
<code>type</code>	[string]	Perform measurements of specified types only. The type equals to "Type" selection in the batch-test definition". Case-insensitive. Default: (all types)
<code>range</code>	[string]	Perform measurements of specified range only. Default: (all ranges)
<code>segment</code>	[string]	Set TP (MX mapping) segment(s) before running the batch-test. Default: (all segments)

Return value

Number of measurements.

Examples

```
#bct | run: resistors
```

Simple run batch-test "resistors".

```
#bct | run: resistors: panel="1,3"
```

Run batch-test "resistors" with panels 1 and 3 only.

```
#bct | run: resistors: range="100k"
```

Run batch-test "resistor" with range "100k" only.

```
#bct | prepare: <name>
```

Prepare-only specified batch-test. Items of specified batch-test will be copied to last-result(s) without performing any measurement. This allows to read-out parameters via result command before run the measurement.

Parameters

<code>name</code>	[string]	Batch-test name to prepare.
-------------------	----------	-----------------------------

Return value

Number of items.

Examples

```
#bct | prepare:resistors
Prepare batch-test "resistors".
```

```
#bct | read?:<name>
```

Read single result value of specified measurement of last batch-test run.

Parameters

name	[string]	Measurement name to read.
------	----------	---------------------------

Return value

Value of measurement.

Examples

```
#bct | read?:R1
Read result value of "R1" measurement.
```

```
#bct | result:cell=[string]{:panel=[list]}{:sheet=[string]}{:format=[string]}
{:offset=[number]}{:max=[number]}
```

Store the whole result of last batch-test run at specified coordinates of specified sheet. Result is stored by lines.

Parameters

cell	[string]	Starting top-left cell. Take
panel	[list]	Filter results to only selected panel(s). Enter numbers, comma separated. Default: (all panels)
sheet	[string]	Optional sheet specification. The sheet must exist. Take care when the "TEST" sheet is used, the command can overwrite your program! Default: BCT
format	[string]	Possibility to define columns to be stored to target cell/sheet. Available columns: <ul style="list-style-type: none"> • enabled • name • type • range • panel • caption • lolimit • lolimit-base (low-limit in base units) • hilimit • hilimit-base (low-limit in base units) • tplow • tphigh • value • value-base (measured value in base units)

		The lolimit, hilimit and value are re-calculated to units, used in the definition. If low-limit is defined, hi-limit and value is recalculated according to lo-limit units. If no low-limit is defined, but hi-limit is, the value is recalculated according to hi-limit units. If both lo-limit and hi-limit are not defined, the value is returned in base units. Enter like a string using column names above, separated by a comma. The order of columns is preserved when storing. Default: " name, value"
offset	[number]	Skip number of results to store. Default: 0
max	[number]	Limit number of results to store. Default: 0 (no limit)

Return value

No return value. The result is directly stored to the specified sheet/cell.

Examples

```
#bct | result:cell="A1"
```

Store name and value results to sheet "BCT", starting by cell "A1".

```
#bct | result:cell="A1":sheet="BCTresult":format="name,lolimit,hilimit,value",max=15
```

Store name, low-limit, hi-limit and value results to sheet "BCTresult", starting by cell "A1". Number of results to store will be limited to 15.

6.4.9.4 #extprocess (Run an external process)

```
#extprocess | start:<filename>{;args=[string]}{;waitforexit=[bool]}
              {;timeout=[number]}{;nowindow=[bool]}
              {;redirstdout=[bool]}{;redirstderr=[bool]}
```

Start a new external process.

Parameters

filename	[string]	Path to an executable file to execute.
args	[string]	Optional command-line arguments.
waitforexit	[bool]	If true, block executing the program until process exit. Default: false
timeout	[number]	Time limit in [ms] to wait for process exit. If the process will not exit in this time limit, the return status becomes 1 (error). Default: (no timeout)
nowindow	[bool]	Hide the window of executing application. Default: true
redirstdout	[bool]	Redirect process standard output ("stdout") stream. If true, the output is received to internal string buffer. This buffer can be accessed by "stdout" command. Default: false
redirstderr	[bool]	Redirect process standard error ("stderr") stream. If true, the error output is received to internal string buffer (use the command "stderr" to access). Default: false

Return value

No return value.

Examples

```
#extprocess | start:"c:\\test.bat"
```

The most simple use, just start the "test.bat" batch file. No arguments. No output(s) redirecting. No waiting for process exit.

```
#extprocess | start:"c:\\test.exe";args="-file abc.txt";  
waitforexit=true;timeout=3000
```

Start the "test.exe" process and pass "-file abc.txt" command-line arguments. Command will block the program execution for a maximum of 3 seconds. If the started process will not finish its work in 3 seconds, the Return Status becomes log.1 (error).

```
#extprocess | quit
```

Quit the running process immediately.

Parameters

No parameters.

Return value

No return value.

```
#extprocess | running?
```

Check if the started process is running.

Parameters

No parameters.

Return value

Returns "1" if the process is running, otherwise "0".

```
#extprocess | retcode
```

Get return code of last exited process.

Parameters

No parameters.

Return value

Typical returns "0" if ok, anything else is an error.

```
#extprocess | stdout:read=[ number]  
#extprocess | stdout:count?  
#extprocess | stdout:clr  
#extprocess | stderr:read=[ number]  
#extprocess | stderr:count?
```



```
#extprocess | stderr: clr
```

Access the internal standard output (stdout) or error output (stderr) buffer. To use "stdout" command, the "redirstdout" parameter of start command must be set to true. To use "stderr" command, the "redirstderr" must be set to true.

Parameters

<code>read</code>	<code>[number]</code>	Line number of internal buffer to read. <ul style="list-style-type: none"> • If positive or equals to a zero (≥ 0), the index points to the start of the buffer. • If negative (< 0), the index points to the end of the buffer. <p>If asterisk (*) is passed, all lines from the buffer, separated by new-line (LF) character is returned.</p>
-------------------	-----------------------	---

Return value

Return value depends on passed command:

- **read=In/-In/*** - return value is the line from the stdout/stderr buffer (or all lines)
- **count?** - return value is line count of the buffer
- **clr** - no return value, clear the buffer

Examples

Following examples expecting filled out the stdout or stderr buffer with for example following lines:

- (0) abc
- (1) def
- (2) ghi
- (3) jkl
- (4) mno

```
#extprocess | stdout: count
Returns 5.
```

```
#extprocess | stdout: read=0
Returns "abc".
```

```
#extprocess | stdout: read=1
Returns "def".
```

```
#extprocess | stdout: read=-1
Returns "mno".
```

```
#extprocess | stdout: read=-2
Returns "jkl".
```

```
#extprocess | stdout: read=*
Returns "abc\ndef\nghi\njkl\nmno".
```

6.4.9.5 #dummy (Dummy-test control)

This command enables to control a dummy-test sequence. Dummy-test is in short the test of the testing fixture. It is based to test and find known errors (dummy components) on FAIL DUTs, which always must be identified correctly. Typical is to perform a dummy-test before the main testing. In fact, the dummy-test is a classic test sequence, but the programmer must take care about notifying dummy component parts during the test.

The dummy-test must be enabled and dummy-test component(s) must be defined in the HEAD sheet of the

test-file.

Dummy-test has its own graphics representation on the operator screen. During dummy-test procedure, the counter section is hidden. Instead of counter section the dummy section is shown, to the bottom corner of the operator screen.

```
#dummy | start:<dummy-test-name>{: enabled=[ bool]}{: passbefore=[ bool]}{:
passafter=[ bool]}
#dummy | reset
```

- Start command (re-)loads the dummy-test components and shows the graphics interface on the operator screen.
- Reset command hides the dummy-test interface and resets the dummy-test.

Parameters

dummy-test-name [string]

In the HEAD sheet, the DUMMY LIST section can be defined more types of dummy tests. Each of them can contain only selected components and specified logic to pass the component test.

enabled [bool]

If no different dummy tests are specified or this argument is omitted, all enabled dummy components are used with 'any' logic (pass or set is used to mark the component as done).

Override the "Enabled" value, written in the HEAD.

Default: "Enabled" value in the HEAD

passbefore [bool]

Override the "PASS before" value, written in the HEAD.

Default: "PASS before" value in the HEAD

passafter [bool]

Override the "PASS after" value, written in the HEAD.

Default: "PASS after" value in the HEAD

Return value

No return value.

```
#dummy | enabled?
#dummy | onbegin?
#dummy | onend?
#dummy | done?
#dummy | started?
```

Get the dummy-test settings (allows to define dummy-test parameters in the HEAD sheet) and the status:

- enabled? - reads the enabled flag in the HEAD definition
- onbegin?, onend? - reads the values from dummy-test definition in the HEAD sheet of the test-file
- done? - reads the status of dummy-test, the dummy-test is done if all defined components already passed
- started? - if dummy-test is started (by a "start" command)

Parameters

No parameters.

Return value

When true, return value is "1".

When false, return value is "0".

```
#dummy | set:<dummy-component>
#dummy | pass:<dummy-component>
```

```
#dummy | fail: <dummy-component>
```

Mark the dummy-test component as done. According declaration in the HEAD sheet, the specified component is marked as done by 'fail' or 'pass' result of specified component on the dummy-sample testing board.

Parameters

`dummy-component` [string]

Dummy-test component name to be marked as passed. The component must be defined in the "Dummy list" in the HEAD sheet.

There are two reserved dummy components (internal, not defined in the test-file HEAD sheet):

- `@before` - if the "PASS before" option is enabled (HEAD sheet), it is required to successfully test the PASS DUT. Until this DUT is placed and tested, it is not possible to mark any other component as passed. The 'pass' command must be used to mark this component.
- `@after` - if the "PASS after" option is enabled (also HEAD sheet), it is required to successfully test the PASS DUT after all previous dummy components are passed. The 'pass' command must be used to mark this component.

Return value

No return value.

Examples

```
#dummy | pass:@before
Mark the PASS before test done.
```

```
#dummy | set:R1
Mark the defined component - "R1" done.
```

```
#dummy | set:C3
Mark another defined component - "C3" done.
```

```
#dummy | fail:L1
Mark another defined component - "L1" done. The meaning is the component L1 must fail to pass the test.
```

```
#dummy | pass:@after
Mark the PASS after test done.
```

```
#dummy | done?
If this commands returns "1", the dummy-test is finished (all components are passed), otherwise not (some of the component is not marked passed).
```

6.4.9.6 #get (Get a value)

```
#get | <source>: <property>{; arg1=[ ?] }{; arg2=[ ?] }...{; argN=[ ?] }
```

Using this function, it is possible to access system properties.

Parameters

`source` [enum] Specify property source.
Valid values:

- system

		<ul style="list-style-type: none"> • project • teststation
property	[enum]	Name of the property. It depends on property source.
arg ₁ ...arg _N	[?]	Optional arguments

Return value

Return value depends on specific source and property. Below you can find all possible source, their properties and return values.

Source	Property	Value type	Description
system	tickcount	[int]	Get a number of milliseconds since operating system start. <i>Note: "tickcount" is stored as a 32-bit signed integer. It will increment to a maximum positive integer value for approximately 24,9 days, then jump to to a minimum negative number and increment back to zero during next 24,9 days.</i>
	loggeduser	[string]	Login of currently logged user in the OS.
	machinename	[string]	Name of the computer.
	winverfull	[string]	Full version of the OS, including name, service pack, etc.
	winver	[string]	Short version of the OS, format a.b.c.d
funtest	netver	[string]	Full name of .NET framework version, used at the compile time.
	ver	[string]	Version of funTEST executable.
	exepath	[string]	Full path to the funTEST's executable.
project	name	[string]	Name of currently loaded project
	dir	[string]	Directory of currently loaded project file
	path	[string]	Full path to currently loaded project file
teststation	name	[string]	Name of currently used teststation
	dir	[string]	Directory of currently used teststation file
	path	[string]	Full path of currently used teststation file
testfile	name	[string]	Name of currently loaded test-file
	path	[string]	Full path of currently loaded test-file <i>Note: when read-only test-files are used, the funTEST returns source path and directory of loaded test-file, not the local working copy</i>
	dir	[string]	Directory of currently loaded test-file

Examples

```
#get | system:tickcount
```

Get a number of milliseconds since OS start, returns for example 45674961, which means the OS is running for about 12 hours, 41 minutes and 15 seconds.

6.5 Statistics

6.5.1 Automated statistics

Automated statistics is represented by "AMSTAT" sheet and dedicated "Automated statistics" column in TEST sheet in the test-file. This module makes statistics much more easier to use comparing to [#stat](#) command.

Principle

On the AMSTAT sheet are defined targets with specified names, which are called by AMSTAT column in the test-file. Each target is defined by target device and command. The target is executed on every line of test-file, where the specified name is used in AMSTAT column.

6.5.1.1 AMSTAT sheet

AUTO STAT section

Enabled - yes or no, global enables or disables the auto stat functionality

Device - the default device to be used, when no device is specified in target definition

Target definitions

- **Name definition** - name, used in AMSTAT column
- **Source column** - name of source column to read the value of current row before execute the command
- **Value** - read value of source column
- **Device** - target device or internal command to use, if empty - the default device is used
- **Command** - command and parameters to be executed

The combination Source column and Value can be used, when reformat of read value from source column to pass to the command is needed. Notice, that this will slow-down the statistics system.

They can be left empty if there is no need to reformat the value.

Command column support **variables** to pass values from test-file row where the define is called:

\$value\$	Pass value, read by Source column
\$step-name\$	Step-name column (test-file)
\$comment\$	Comment column (test-file)
\$judge\$	Judge column (test-file), formatted like "0" (OK) or "1" (NG)
\$!judge\$	Inverted judge column (test-file), "1" if OK, otherwise "0"
\$target-value\$	Target value column (test-file)
\$unit\$	Unit column (test-file)
\$lo-limit\$	Lo limit column (test-file)
\$hi-limit\$	Hi limit column (test-file)
\$result\$	Result column (test-file)
\$return-value\$	Return value column (test-file)
\$step-nr\$	Row number of caller step (equals to spreadsheet row number)