TEST-OK

# Test Language Description

Version 3.4.0.0

7/11/2014

# Table of contents

# Overview

This document describes the TEST-OK Description Language (TDL) that allows the user to create test sequences that will test the Unit Under Test (UUT) or PCB that is connected to the TEST-OK System.

Note that it is not a manual for the use of TEST-TRACK, My-TEST or eC-my-test, the TEST-OK applications that are used to control the sequencing of one or more tests. Nothing is therefore said in the next chapters about relations between test suites, scripts, preambles, partly re-running of test suites etc. Please refer to the TEST-TRACK, My-TEST or eC-my-test User Manual for more information.

Some commands are not available in My-TEST and eC-my-test. Where applicable, this is indicated.

One particular detail may however be useful to mention here: at the beginning of a test suite run (consisting of one or more tests), all functional modules of the Test Controller Card or TEST-Mate(s) that are controlled are reset, meaning all outputs are off and all configurations are reset. At the end of each test, the outputs are left as they are, only at the end of the last test are all outputs reset again. In order to set all outputs in a known state at the start of a test (independently of other tests having been executed or not), it is advised to use the 'preamble' option.

# Language overview

The test language is meant to conduct tests with a TEST-OK fixture with a Test Controller Card or one or more TEST-Mates, testing assembled PCBs or even complete products. Each test consists of a number of commands that are executed by the Test Engine, which is part of the software that is used to conduct a test.
The test language allows the user to apply stimuli to the PCB (or UUT, Unit Under Test as it will be called from here on) and to test measured values on the UUT against expected values.
Most commands of the Test Description Language (TDL for short) results in either a stimulus to be sent to the UUT or a state to be read. The reading of a state can be accompanied with a condition. If the condition is false, the test verdict is set to failed.
This method is fine for simple tests on single inputs but it will not work if the verdict is based on several inputs or on changes in an input over time. For these more complex conditions TDL supports variables and arithmetic operations on these variables. In combination with the IF and WHILE commands, it is possible to create any condition necessary to create the test verdict.

Another type of commands controls the test flow, e.g. repeating a set of commands for different values or waiting until a specific condition occurs.

The inputs and outputs provided by the TEST-OK test controllers are numbered (e.g. analog output 1..16 or 201..202, or digital input 1..24). When writing a test, the test author will rather think of the UUT's functions and signal names instead of the numbered I/O. A special command (called MAP) is provided to assign a name to an input or output or, in case of digital input or outputs, to a group of inputs or outputs. Using these names in I/O commands improves readability.

A LOG command is provided to write any text to the log output window. Using some special character sequences allows to include e.g. the values of any variable or the latest output value or input value to be included in the text.

Nowadays, many PCBs contain a microcontroller. This makes it very hard to test a number of functions by simply applying stimuli to the board and reading back some TEST-OK inputs. It is for instance impossible to test whether a real-time clock device (RTC) on the PCB is working correctly without reading one of its status registers.
A common solution is the use of a communication channel of the microcontroller over which a special 'test mode' packet is sent. The microcontroller will use such a packet to send for instance the RTC status register.
Depending on the type of Test Controller, several standard interfaces are supported, UART, RS485, CAN, I2C to name a few. The TDL supports these interfaces by means of commands that allow sending and receiving of user-defined packets.

In order to add the greatest possible flexibility, it is also possible to interface with external applications in different ways: running an application while passing parameters via a command line interface, communicating with a TCP server application or VISA for control of test and measurement equipment.

Finally, TDL provides support for programming micro controllers during the test session[1]. This avoids extra handling of the UUT since the same fixture is used for both programming and testing.
Currently, Microchip PIC controllers and the Philips LPC2k series are supported. Check the TEST-OK support to find out about other processors.

The next sections give an overview of the type of commands that are available (2.1), the use of variables (2.2), formatting of strings (2.4) and serial communication with the UUT (2.5).

1 This requires that the tester hardware has programming capabilities

## Commands

The next subsections shortly describe the available commands. Refer to the Detailed Command Description for a detailed description.

Note that the commands can be written in upper case, lower case or a mix of the two.
Many commands have optional parameters which can be omitted when not relevant or when the default value of that parameter is used.
The order of the parameters is important, they must be specified in the order described in this document.
Variables and other names are case sensitive, i.e. the variable #MyVar is not the same variable as #MYVAR.

## Stimuli

These commands control the outputs of a TEST-OK Test Controller Card. This may either be directly as for power supplies, digital outputs and analog outputs or indirectly for more complex functions such as the In-Circuit Programmers or the values of packets that are transmitted via a serial interface to the UUT.

| ***Command*** | ***Parameters*** | ***Description*** |
|---|---|---|
| Set_Supply | Channel, OnOff | Controls the specified power supply |
| Set_Digital | Group + Bit or Name, OnOff | Sets/resets the specified digital output |
| Set_Analog | Channel, Value | Set the specified analog output to the specified value |
| Set_Pwm* | Channel, OnOff | Controls the specified PWM channel |
| Program* | Channel, FileName | Uses the specified programmer to program the hex file into the UUT |
| Reset_Counter | Channel | Resets the specified counter |
| Set_SerialField* | Channel, FieldName, Value | Modifies the specified field of the packet that is sent over the specified serial channel to the UUT. |
| Transmit_Serial | Channel | When the serial channel is configured in 'manual' mode, this command will send one packet to the UART on the Test Controller Card. |
| Transmit_CAN | Channel | This command will send one packet to the UUT via a CAN interface on the Test Controller Card. |
| Transmit_RS485 | Channel | This command will send one packet to the UUT via an RS485 interface on the Test Controller Card. |
| Transmit_Com* | Channel | Sends an ASCII message to a COM port on the PC |
| Transmit_Tcp* | Channel | Sends an ASCII message to an external application |
| Transmit_Visa* | Channel | Sends an ASCII message to a VISA instrument |
| I2C_Start | Channel | Applies the $I^2C$ START condition to the specified $I^2C$ interface |
| I2C_Stop | Channel | Applies the $I^2C$ STOP condition to the specified $I^2C$ interface |
| I2C_Write | Channel, Data | Writes one byte of data to the specified $I^2C$ interface |
| I2C_Read | Channel | Reads one byte of data from the specified $I^2C$ interface |

*Not available in My-TEST and eC-my-test

## Testing

Test commands deal with inputs of the TEST-OK tester. This can be analog or digital inputs or more complex functions such as the current drawn from a power supply or a counter.

The general format is:

```
test_cmd <item to test> EXPECT <condition> ELSE <condition mode> 'message'
```

All test commands have a condition part, a condition mode and a text string (the only exceptions are the RECEIVE_xxx and CALIBRATE commands).
A condition is a simple comparison of the retrieved value against an expected value. Comparisons are: <, >, <=, >=, == (equal to) , != (not equal to) and 'always true' (i.e. no condition).
Comparisons can be combined to more complex conditions using AND, OR and NOT plus parentheses.

If the condition fails, then the condition mode indicates how the remainder of the test shall be handled:

| ***Condition Mode*** | ***Description*** |
|---|---|
| IGNORE | The error will be ignored and execution continues as if the command was executed without an error. The verdict of the current test is not altered.<br>No error messages will be logged. |
| CONTINUE | The current test shall simply continue, followed by the execution of any subsequent tests. The verdict of the current test is set to FAILED. |
| ABORT | The current test shall be aborted while subsequent tests shall still be executed. The verdict of the current test is set to FAILED. |
| ABORT_ALL | The current test shall be aborted immediately without executing any other tests. The verdict of the current test is set to FAILED. |

If the condition fails, the specified text string is shown on the log screen.

Test commands can also be used to assign a value to a variable, allowing more complex conditions to be constructed with the IF command. When using a test command for assignment, it is not necessary to include the 'EXPECT <condition> <mode> “message”' part although this is still allowed (in the latter case, the measured value is assigned to the variable and the condition is evaluated in the normal way).

| ***Command*** | ***Parameters*** | ***Description*** |
|---|---|---|
| Test_SupplyCurrent | Channel, Current | Tests the supply's current against an expected value |
| Test_Digital | Bit or Name | Tests the specified input(s) against an expected value |
| Test_Analog | Channel | Tests the specified input against an expected value |
| Test_Counter | Channel | Tests the specified counter against an expected value |
| Test_Time | - | Compares the system time with the expected value defined in the condition part |
| Test_SerialField* | Channel, Fieldname | Test the specified field of the last received packet from the UUT against an expected value |

*Not available in My-TEST and eC-my-test

Additional commands that are 'testing' are listed in the table below, they do not have the same structure as the TEST_xxx commands above. These are mainly receivers for the various communication interfaces plus the 'calibration' command.

| ***Command*** | ***Parameters*** | ***Description*** |
|---|---|---|
| Receive_Serial | Channel, Expected packet structure | Tests whether the received packet from the UART on the Test Controller Card is conform to what is expected |
| Receive_CAN | Channel, Expected packet structure, Expected Id | Tests whether the received packet on the CAN interface on the Test Controller Card is conform to what is expected |
| Receive_RS485 | Channel, Expected packet structure | Tests whether the received packet on the RS485 interface on the Test Controller Card is conform to what is expected |
| Receive_Com* | Channel, Expected packet structure | Tests whether the packet received from a COM port on the PC is conform to what is expected |
| Receive_Tcp* | Channel, Expected packet structure | Tests whether the packet received from an external application is conform to what is expected |
| Receive_Visa* | Channel, Expected packet structure | Tests whether the packet received from a VISA instrument is conform to what is expected |
| Calibrate | Channel, { min .. max}, Caption | Shows a dialog box with the real-time value of the specified analog input channel. A correctly calibrated value shall lie within Min and Max. If the OK button is pressed, the actual value is checked on the valid range and determines whether the command passes or fails |
| Overload | Channel, Current | Reads the supply's current overload status |

*Not available in My-TEST and eC-my-test

## Flow Control

Some elementary flow control commands allow to either wait for an expected event or to repeat a sequence of commands several times with different values.

| ***Command*** | ***Parameters*** | ***Description*** |
|---|---|---|
| WaitMs | Timeout | Waits for the specified number of ms |
| WaitWhile | Timeout, Message | Continues executing the test command immediately following this command as long as this test is true or until we reach the timeout.<br>Note: when the test command condition fails, the loop is exited but the test verdict is NOT set. |
| For | Variable { values } | Repeats the commands until the EndFor for all variable values.<br>Values are either an enumerated set of values or a range |
| While | Condition | Repeats one or more commands as long as the specified condition is met |
| If | Condition | Allows to execute a list of commands only if the specified condition is met or is not met |
| WaitTx* | Channel, Timeout | Waits until a new serial packet has been transmitted on the specified channel |
| WaitRx* | Channel, Timeout | Waits until a new serial packet has been received on the specified channel |
| Fail | Message | Makes the test fail. This command is useful in IF Statements. It allows to make a test fail on much more complex conditions than the simple TEST_xxx commands. |

*Not available in My-TEST and eC-my-test

## Configuration

The configuration commands will configure the test hardware.

| ***Command*** | ***Parameters*** | ***Description*** |
|---|---|---|
| Map | Name, InOut, Group, [Bits....] | Groups one or more digital inputs or outputs into a new name |
| Config_Serial | Channel, BaudRate, Parity, TxMode, RxMode, RxThreshold, RxTimeout, StxMode, EOL character(s) | Configures the specified serial channel on the Test Controller Card |
| Config_CAN | Channel, BaudRate, Use of extended ID or not | Configures the specified CAN channel on the Test Controller Card |
| Config_RS485 | Channel, BaudRate, Parity, RxThreshold, RxTimeout, EOL character(s) | Configures the specified RS485 channel on the Test Controller Card |
| Config_Digital_Group | Group, Voltage | Sets the '1' level output voltage for the specified group (note that specifying the voltage for open collector/ open drain outputs will have no effect) |
| Config_Pwm* | Channel, Frequency, Duty Cycle | Sets the frequency and duty cycle of the specified PWM channel |
| Config_Counter* | Mode | Sets the counter to Count or Frequency mode |
| Config_Supply | Channel, Voltage, MaxCurrent | Sets the voltage and current limit of the specified supply |
| Config_Programmer* | Channel, DeviceName, Mode, Supply, Speed | Configures the specified programmer for a specific device |
| Config_Com* | Channel, BaudRate, Parity, EOL character(s) | Configures and opens a COM port on the local PC |
| Config_Tcp* | Host, Port, EOL character(s) | Configures and opens a TCP connection |
| Config_Visa* | Instrument | Configures and opens a connection to a VISA instrument |

*Not available in My-TEST and eC-my-test

## User Interaction

Interaction with the user is sometimes necessary: the user has to give feedback on whether e.g. a LED is on or the user needs to turn a potentiometer in order to calibrate a circuit.

| ***Command*** | ***Parameters*** | ***Description*** |
|---|---|---|
| Ask | Message, PictureFile, Type | Opens a dialog with the user, showing the message (or question) and expects either 'OK' or a 'YES/NO' answer, depending on the specified Type. Optionally, a picture can be shown below the text. |
| Show_Message, Hide_Message | Message, PictureFile, | Opens a message box with the specified message and, optionally, a picture. There is no feedback from the user expected and the message box is removed by the script with a Hide_Message command. |
| Play_Sound | FileName, Mode | Plays the specified sound file. The command either waits on the completion of the playing or not |
| Calibrate[2] | Channel, { min .. max}, Caption | Shows a dialog box with the real-time value of the specified analog input channel. A correctly calibrated value shall lie within Min and Max. If the OK button is pressed, the actual value is checked on the valid range and determines whether the command passes or fails |

2 This command is the same as is listed in the 'Testing' section above

## Miscellaneous

These commands do not fit in any of the above categories.....

| ***Command*** | ***Parameters*** | ***Description*** |
|---|---|---|
| Var | Variable Name, [Default Value] | Defines a variable to which input values or expressions can be assigned. |
| Log | Message, Type, Indent | Shows the specified message on the log screen |
| Run | Command line | Executes an external program |
| Disable_BoardDetect | - | Disables the board detect function that stops any running test when removing the UUT |
| Enable_BoardDetect | - | (re)Enables the board detect function that stops any running test when removing the UUT |
| Repair* | Problem description, problem class | Adds the specified repair item to the currently tested UUT |

*Not available in My-TEST and eC-my-test

## Failure Conditions

Besides the input commands, other (implicit) error conditions that make the test fail exist. Examples of failure are:

- Failure of configuration
- Overload of one of the power supplies
- Programming errors
- Tester not ready
- Removal of UUT
- .....

## Timing and Synchronization

The test interpreter uses a number of rules in order to guarantee the correct working of the tester. There is a trade off between execution speed and reliability: should the test software wait for an acknowledge of the test hardware after each command, slowing things down, or should the test software simply issue commands to the hardware without bothering whether the command has been executed before issuing a new command?

For configuration commands, it is best to delay the next command until the configuration has been done. However, if a number of outputs must be set prior to the testing of an input, it is probably best not to wait in between every set command but simply assure not to start the test command before the last set command has been acknowledged.

If several test commands follow each other without any output command in between, then these test commands will all be executed immediately without waiting for new input data from the Test Controller Card before each test command.

The synchronization mentioned here is done implicitly by the test software. Other synchronization needs, specific to the UUT, may exist. Examples are for instance the start-up time of a circuit on the UUT. When the control input of that circuit is activated, then it may take some time before e.g. a corresponding voltage has risen to its required value. In such a case, it is up to the user to take this into account in the test script by using e.g. a WAITMS command or a WAITWHILE command.

The synchronization rules used internally are:

- Configuration commands shall wait until the hardware has acknowledged the termination of the required action(s)
- Test commands will not start until all preceding commands are executed by the tester hardware

## Variables

Variables can be used to store measured values or results of expressions. Before using a variable, it must be declared with the VAR command. Distinction is made between global variables and local variables: global variables keep their values during the entire test suite while local variables have a lifetime of only the current script. See the description of VAR for further details.

**Until a value has been assigned, the value of a variable is 0 or “0” depending on whether the variable is used in an integer of string context (note that the default string value may be surprising since in most languages it is an empty string “”).**

## Variable Types

Variables can have an integer value or a string value. There is no type distinction when a variable is declared (with the VAR command) and its actual value depends on what was last assigned. The type of the value depends on the context in which the variable is used and the command interpreter will try to convert the last assigned value to the type that is most suitable. Three different cases can be distinguished:

1. All variables and constant values in the context are integer type
2. All variables and constant values in the context are string type
3. Variables and constant values in the context have different types

The first two cases are straightforward since they involve only one type and no conversion is required. The third case requires conversion of integer types to string types or vice verse. Which conversion is done depends on the operator that applies to the two types.

Some examples may clarify this:

### Integer type only

A variable that has been initialized with an integer has an integer value.
When this variable is used in an expression that only contains integers, then the variable is used as an integer type with value 123 and the result is easy to understand:

```
VAR #MyVar = 123;
VAR #Result;
#Result = #MyVar * 10  + 4;
```

The variable 'Result' will have a value of 123 * 10 + 4 = 1234.

### String type only

There are only a limited number of cases in which a variable will 'see' other string type variables or constants. Typical examples are comparison operators (in IF and WHILE statements) and the concatenation operator ('+'):

```
VAR #ComPort;
IF ( #_HOST_NAME_ == "Computer_1" )
    #ComPort = 3;
ELSE
    #ComPort = 27;
ENDIF;
CONFIG_COM[#ComPort] BAUDRATE = 19200, EOL = CRLF;
```

In the above example, the COM port that is opened depends on the computer on which the script is run (#_HOST_NAME_ is a string type built-in variable that holds the name of the PC on which the script is running).

Concatenation is done with the '+' sign. The following example will result in 'Hello world' being printed.

```
VAR #Start = "Hello";
VAR #End   = "world";
VAR #Greeting;

#Greeting = #Start + " " + #End;
```

```
LOG "#Greeting#";
```

**Mixed types and implicit or explicit conversion**

If an operator is applied to two operand of different types (one integer, one string), then the integer operand is implicitly converted to a string before the operator is applied.

Es an example, an integer type variable used in an IF-statement that compares it to a string (e.g. the counter part of the serial number of the UUT), will result in the conversion of the variable's (integer) value to a string before it compares the two.
Supposing the serial number of the UUT is “123”, the following IF-statement will succeed and the LOG command will be executed (see Built-in Variables below for an explanation of the #_SN_COUNTER_ variable).

```
VAR #MyVar = 123;                          // integer value
IF ( #_SN_COUNTER_ == #MyVar )          // MyVar is compared to a string
LOG "The serial number matches with MyVar";
ENDIF;
```

For comparisons, the implicit conversion will rarely give a problem but for assignments errors (e.g. impossible to divide two strings) or unexpected results may occur as shown in the example below where 10 is converted to a string since the other operand (the variable #OneThousand) is a string:

```
VAR #Hello = "Hello world";
VAR #OneThousand = "1000";
VAR #Result;

//--- Error message "The '/' operator cannot be applied to strings"
#Result = #OneThousand / 10;
```

In order to avoid such situations, it is possible to override the automatic conversion of variable values by specifying INT or STRING in front of the concerned variable:

```
#Result = INT #OneThousand / 10;
```

INT or STRING apply only to the first operand to the right of it (like the minus sign). For better readability, it is advised to add parenthesis:

```
#Result = (INT #OneThousand) / 10;
```

A more complicated example in which first two strings are concatenated before the result is multiplied by two is given below:

```
VAR #S1 = "5";
VAR #S2 = "1";
VAR #Result;

#Result = INT (#S1 + #S2 + "0") * 2;
```

#S1 + #S2 + “0” results in “510”. The INT converts this to the integer value 510 which is then multiplied by 2. The result is therefore 1020;

Conversion from integer to string is always possible. The other way around is not guaranteed. “1000” will of course result in 1000 but “hello world” cannot be converted to an integer value and will result in an error message:

```
VAR #Hello = "Hello world";
#Result = (INT #Hello) / 10;                  // [1] Error message
```

Finally, an example in which two integers are OR-ed and where the result is stored as a string:

```
VAR #Result;

#Result = STRING ( 0x33 | 0x0f );
```

The logical OR of 0x33 and 0x0F results in 0x3F which is 63 in decimal. The variable #Result will therefore get the value “63”.

## Arrays

Array variables can be defined by adding '[]' at the end of the variable name in the VAR command. There is no need to explicitly specify the size and the type of the array. The size is automatically adjusted as a function of the highest used index in assignments. Trying to get a value with an index above the current size will cause an error message. The type of an array is determined by the type of the first initialization. **All other assignments must have the same type.**

Valid index values are integers starting at zero.

Examples:

```
VAR #Array[];
VAR #PredefArray1[] = { 1000, 1010, 1020, 1030, 1040 };
VAR #PredefArray2[] = { 1000 .. 1005 };
VAR #PredefArray3[] = { 1000 .. 2000 STEP 100 };    // 1000, 1100 .. 1900,
2000
VAR #FrenchNumbers[] = { "zero", "un", "deux", "trois", "quatre", "cinq" };
```

After these lines, #Array has no size yet, #PredefArray1 has a size of 5, with #PredefArray1[0] being 1000 and #PredefArray1[4] being 1040. Instead of enumerating the values, it is also possible to specify a range as is shown for #PredefArray2 and #PredefArray3 in the example above.
In the above examples, initialization was done with constant values. It is however also possible to use expressions which allows more dynamic initialization depending on e.g. earlier results:

```
VAR #Offset;
#Offset = TEST_ANALOG[1];
VAR #ExpectedValues[] = { #Offset + 1000, #Offset + 1010, 2 * (#Offset + 4)
};
```

## Built-in Variables

There is also a set of built-in variables (all names starting with '#_" and ending with '_'). These are variables that need not to be defined with a VAR command and whose value is set by the TestEngine. In order to avoid confusion, it is advised not to use an underscore at the start of user-defined variables.

Below are tables that list all built-in variables.

### UUT Specific

These variables are specific to the currently tested UUT and are only meaningful if testing with a serial number ( i.e. #_SERIAL_NUMBER_ is not equal to -1).

**When using My-TEST or eC-my-test, these variables are not set.**

| *Name* | *Type* | *Description* |
|---|---|---|
| #_HEX_FILE_ | String | The name of the hex file passed by the application which is to be used by the programmer.<br><br>Note: TEST-TRACK derives the value from the 'sw version' <file name> that is specified for the current order line as follows:<br>1. <file name> contains a path (i.e. at least one slash in the name)<br>#_HEX_FILE_ = <file name><br>2. <file name> has an extension but no path:<br>#_HEX_FILE_ = #_SCRIPT_DIR_\ HexFiles\<file name><br>3. <file name> has no extension and no path:<br>#_HEX_FILE_ = #_SCRIPT_DIR_\ HexFiles\<file name>.hex |
| #_ORDER_NUMBER_ | String | The currently selected order number on which the UUT is tested.<br>If running without a serial number, the value is “” |
| #_ORDER_LINE_ | String | The currently selected order line on which the UUT is tested.<br>If running without a serial number, the value is “” |
| #_ORDER_QTY_ | Integer | The order quantity of the currently selected order line. If running without a serial number, the value is 0 |
| #_SERIAL_NUMBER_ | String | The serial number of the board that is currently tested. If no number is associated with the test run, then the value is set to -1.<br>Assigning a serial number that contains non-digit characters to an integer variable results in a value of -2.<br>When assigning to an integer variable, the resulting value is:<br>· -1 if no number is associated with the test run<br>· -2 if the number contains non-digit characters<br>· the value of the serial number |
| #_SN_PREFIX_ | String | The prefix part of the serial number of the board that is currently tested. If no number is associated with the test run, then the value is an empty string.<br>When used to assign to an integer variable, see #_SERIAL_NUMBER_. |
| #_SN_COUNTER_ | String | The prefix part of the serial number of the board that is currently tested. If no number is associated with the test run, then the value is an empty string.<br>When used to assign to an integer variable, see #_SERIAL_NUMBER_. |
| #_SN_SUFFIX_ | String | The prefix part of the serial number of the board that is currently tested. If no number is associated with the test run, then the value is an empty string.<br>When used to assign to an integer variable, see #_SERIAL_NUMBER_. |
| #_UUT_TYPE_ | String | The type of the UUT (as specified in the 'Board Type' Adminstration of TEST-TRACK) |

### Test Sequencing

These variables indicate the progress of the test session and can be used to decide e.g. whether this is the first or last test or the first or last UUT in a group or panel. Using this information it is for instance possible

to execute (part of) a script only when this is the last test or the first UUT in a panel.

| Name | Type | Description |
|---|---|---|
| #_BOARD_POSITION_ | Integer | In case the board is part of a panel or a group, this variable indicates the physical position of the board in the panel or group. The first board that is tested has position 1.<br>If there is no panel or group, the position is 0 |
| #_NUM_OF_UUTS_ | Integer | Number of Units Under Test that are tested in the current run. In case the UUT is part of a panel or a group, the number of UUTS may be more than 1. Note: this variable indicates the ACTUAL number of UUTs being tested. If rerunning only failed UUTs or when one or more UUT is disabled, the numer of of UUTs will be less than the total number of UUTs in the panel or group |
| #_UUT_NR_ | Integer | Current UUT sequence number. Always starts at 1 and may increment up to #_NUM_OF_UUTS_.<br>Do not confuse with #_BOARD_POSITION_: if e.g the first UUT is skipped, #_BOARD_POSITION_ will start at 2 while #_UUT_NR_ will start at 1 |
| #_NUM_OF_TESTS_ | Integer | Total number of tests in the currently executed test suite |
| #_TEST_NR_ | Integer | Sequential number of the test. The first executed test in a test suite has number 1. |

### Test Execution

Concerns the result of the last executed test or test script.

| Name | Type | Description |
|---|---|---|
| #_ERROR_ | Integer | Status of the last executed testing command.<br>See #_ERROR_ : Handling errors programmatically for more information. |
| #_IN_ | Integer | Input value as retrieved from the last TEST_ command or some other commands (as mentioned in their description) |
| #_OUT_ | Integer | Output value as set by the last SET_ command |
| #_PREVIOUS_TESTS_PASSED_ | Integer | Indicates whether all previous tests in the current test session passed (value = 1) or whether one or more tests failed (value = 0).<br>This can be used to make actions depend on results of already executed tests. |
| #_SCRIPT_DIR_ | String | Name of the directory in which the currently executed script is located. |
| #_TEST_FAILED_ | Integer | Becomes non-zero after the first encountered error in the current script.<br>Contains the last assigned value of #_ERROR_.<br>See section 4.2.1, '#_ERROR_ : Handling errors programmatically' for more information. |
| #_WAITED_ | Integer | The time spent in the last WAITWHILE command |

### Date and Time

Variables that indicate the current date and time

| Name | Type | Description |
|---|---|---|
| #_YEAR_ | Integer | The year of the current date (e.g. 2012) |
| #_MONTH_ | Integer | The month of the current date (1 .. 12) |
| #_DAY_ | Integer | The day of the current date (1..31) |
| #_HOUR_ | Integer | The hour part of the current time |
| #_MINUTE_ | Integer | The minutes part of the current time |

### Command Specific

These variables are set or read by specific commands. Refer to the associated command for any details.

| *Name* | *Type* | *Description* |
|---|---|---|
| #_CAN_RX_ID_ | Integer | ID of last received CAN packet. See RECEIVE_CAN command. |
| #_CAN_RX_DLC_ | Integer | Length of the received packet. See RECEIVE_CAN command. |
| #_CAN_RX_DATA_ | Integer array | Data byte of the received packet. See RECEIVE_CAN command. |
| #_CAN_STATUS_ | Integer | Error flags for CAN bus interface. See RECEIVE_CAN command for the definition of the various flags. |

### Miscellaneous

Information about the test environment. Note that most of these variables are only available when using TEST-TRACK.

| **Name** | **Type** | **Description** |
|---|---|---|
| #_COMPANY_NAME_ | String | Full name of the company of the currently logged-in operator |
| #_COMPANY_CODE_ | String | Code (abbreviation) of the company of the currently logged-in operator |
| #_EXPANSION_BOARD_ID_ | Integer | The ID of the expansion board that is present in the fixture. |
| #_HOST_NAME_ | String | The name of the PC on which the test is running |
| #_MODULE_ID_ | Integer | The ID of the TEST-OK module on which the board is tested |
| #_OPERATOR_NAME_ | String | Full name of the currently logged-in operator |
| #_OPERATOR_CODE_ | String | Code (abbreviation) of the currently logged-in operator |
| #_TCC_ID_ | Integer | The serial number of the Test Controller Card (TCC) that is currently connected |
| #_TCC_TYPE_ | String | The type of TCC that is currently connected (e.g. "TCC1800USB") |

## Functions

The Test Description Language provides a set of helpful built-in functions that can be used in expressions. The next subsections describe them in detail. The convention used to indicate return type and arguments is as follows:

```
<return type> <functionname> ( <argument type> <argument name> , more arguments )
```

The return type can be either an integer value (INT) or a string (STRING). Argument types are also INT or STRING.

### Mathematical

#### abs

```
INT abs ( INT value )
```

Returns the absolute value of the specified value.

#### min

```
INT min ( INT value1, INT value2 )
```

Returns the minimum of the two specified values.

#### max

```
INT max ( INT value1, INT value2 )
```

Returns the maximum of the two specified values.

### String Manipulation

#### strlen

```
INT strlen ( STRING String )
```

Returns the length of the specified string.

#### substring

```
STRING substring ( STRING String, INT Index, INT Count )
```

Returns a substring of the specified string, containing count characters beginning at index.
The first character in the string has index 1. If the specified Index is less than 1, it is silently supposed to be 1. For Index and Count values beyond the length of the string, an empty string ("") is returned.

#### trim

```
STRING trim ( STRING String )
```

Returns a string that is obtained by removing leading and trailing spaces and control characters from the specified string. Use trim to remove blank spaces before and after the first printing character.

#### trimleft

```
STRING trimleft ( STRING String )
```

Returns a string that is obtained by removing leading spaces and control characters from the specified string. Use trimleft to remove blank spaces before the first printing character.

#### trimright

```
STRING trimright ( STRING String )
```

Trim returns a string that is obtained by removing trailing spaces and control characters from the specified string. Use trimright to remove blank spaces after the first printing character.

### lowercase

```
STRING lowercase ( STRING String )
```

Returns a string that contains all characters in this string, converted to lowercase.

### uppercase

```
STRING uppercase ( STRING String )
```

Returns a string that contains all characters in this string, converted to uppercase.

### evaluate

```
STRING evaluate ( STRING String )
```

Performs a variable substitution as described in section 2.4.1 on the specified string and returns the result. Example:

```
VAR #VersionH = 2;
VAR #VersionL = 9;
VAR #Version1;
VAR #Version2;

#Version1 = "#VersionH#.#VersionL:02d#";
#Version2 = evaluate( "#VersionH#.#VersionL:02d#" );
```

Normally, variable substitution is not applied to strings that are part of an expression (like the assignment to a variable). After execution of the above example, the value of #Version1 is therefore "#VersionH#.#VersionL:02d#". The evaluate() function will substitute any #-delimited variable name with the actual value of the variable. #Version2 will therefore be "2.9".

# String Formatting

## Values in Output Strings

Output strings are strings that are either printed to the log screen or are transmitted to other devices (e.g. the serial channel of the Test Controller Card or to TCP). Strings that are assigned to a variable are also considered as 'output'.

Besides fixed text, it is useful to be able to also print e.g. the last measured value or the result of a calculation involving several samples.
In order to print such values, a mechanism is provided that substitutes specially formatted character sequences with an actual value:

> Values of variables can be printed by simply using the name of the variable enclosed between '#' characters[3]. When the message is printed, any occurrence of '#<name>#' (where <name> is a variable name) is substituted by the current value of that variable. An error is issued if <name> is not a defined variable.

An optional format specifier can be added to force printing the value in decimal, floating point, hexadecimal or binary format. The specifier is added between the variable name and the closing '#' in the following way:

```
#<name>:<size><format>#
```

The <size> part is optional while the <format> must always be present. The <size> specifies the minimum size of the printed value. If the number of digits in the value representation is less than this minimum size, then the actions described in the table below will be done.
If the <format> is not specified, then integer type values will be printed in decimal form and string type variables as strings. If <format> is specified and the value is a string, then the TestEngine will try to convert the string to an integer. If this is not possible, then 0 will be printed. (i.e. if the string value is “123”, then printing with the 'x' format specifier will result in 7B to be printed, if the string is “hello”, then 0 will be printed)

| **<format>** | **Representation** | **Effect of <size> if the number representation has less then <size> digits** |
|---|---|---|
| d | Decimal | Spaces are added in front of the number. If <size> starts with a '0', the leading zeroes are added instead of spaces. |
| f | Floating point. The value is divided by 1000 and represented with a decimal point<br>A floating point number is a number with a period '.' and an optional exponent part: <i>.<f>E<exp> | <size> indicates the number of digits right from the decimal point. |
| x | Hexadecimal | Zeros are added in front of the number |
| b | Binary | Zeros are added in front of the number |
| r<n><e> | Raw byte values. Allows to insert non-ASCII values.<br>n: Number of bytes that must be used for the value. Default value is 1.<br>e: 'Endianess': 'l' stands for LSByte first, 'm' stands for MSByte first. Default value is MSByte first. | The size (the number of bytes that is inserted) is always equal to <n> |

3 Any literal '#' that is not associated with a variable must be preceded with a '\'. See 2.4.3

Variable values are always integer values. Variables that represent a voltage or current are always in mV or mA. In order to print these values in V or A, the floating point specifier can be used.

**The substitution mechanism is valid for all strings in all commands but not for expressions (e.g. in variable assignments[4]).**

Some examples:

Testing an analog input results the 'last input value' being updated. Using #_IN_# in the error message of that TEST_ANALOG command will show the value in the log window.
The result of reading the analog input is also assigned to a variable which can then be used in e.g. a LOG command as shown below.

```
VAR #Input;

#Input = TEST_ANALOG [ $V_out ]
           EXPECT (> 2.0 AND < 4.0),"Voltage out of range (#_IN_:f# V)";

LOG "Integrator voltage at t=1sec: #Input# mV", INDENT = 1;
```

Instead, one may also use:

```
LOG "Integrator voltage at t=1sec: #_IN_# mV", INDENT = 1;
```

because the value tested by the TEST_ANALOG is also the last read input value.

In the next example, we control a bar of LEDs via a 3-bit digital output bus. For every level, we ask the user whether the correct number of LEDs are on.

```
VAR #Reply;

MAP $LedCtrl ON DIGITAL OUT GROUP 201, BIT 1..3;

FOR #Level { 0 .. 7 }
    SET_DIGITAL [ $ LedCtrl ] = #Level;
    #Reply = ASK "Are the first #Level#" LEDs on?", Type = YESNO;
    IF ( #Reply == NO )
          FAIL "Led #Level# is not on";
    ENDIF;
ENDFOR;
```

4 See 2.3.2.8 on how to force variable substitution in expressions

Formatting examples:

```
VAR #Bin = 25;
VAR #Hex = 0x1A33;
VAR #Voltage = 2456;
VAR #Number = 100;
VAR #VersionH = 2;
VAR #VersionL = 9;

LOG "Bin is 0b#Bin:b# or 0b#Bin:8b# or 0x#Bin:x# or #Bin:d# in decimal";
LOG "Hex is 0b#Hex:b# or 0x#Hex:x# or #Hex# mV or #Hex:f# V";
Log "Voltage is #Voltage# mV or #Voltage:f# V";
Log "Number is #Number:2f# or 0x#Number:4x#";
log "Version #VersionH#.#VersionL:02d#";
```

The corresponding output on the screen will be:

```
[Info   ] Bin is 0b11001 or 0b00011001 or 0x19 or 25 in decimal
[Info   ] Hex is 0b1101000110011 or 0x1A33 or 6707 mV or 6.707 V
[Info   ] Voltage is 2456 mV or 2.456 V
[Info   ] Number is 0.10 or 0x0064
[Info   ] Version 2.09
```

The substitution mechanism also works for strings in the TRANSMIT_xxx commands. The 'r' format specifier works only for transmission commands and allows, together with the '\xnn' notation (see 2.4.3 below), to construct binary (i.e. non-ASCII) packets.
As an example, sending a four byte packet that contains a two-byte value (LSByte first) that starts with 0x03, ends with 0x02 can be sent like this:

```
VAR Value = 0xABCD;
TRANSMIT_COM[2] "\x03#Val:r2l#\x02";
```

The actually sent bytes will be 0x03, 0xCD, 0xAB, 0x02  (plus eventually CR and/or LF depending on the configuration of the channel)

## Values in Input Matching Strings

Input matching strings are strings that specify how another string (usually received from the UUT, an external application or measurement instrument) is expected to be. Special 'format specifiers' can be used to indicate that a part of the string is not invariable.
For those familiar with C/C++, it is the equivalent of the format string in scanf().

### Wildcards

Wildcards are special characters can be used to substitute for one or more characters in the string. Two different wildcards are defined:

- The Asterisk character ('*') matches any received string.
  This wildcard can only be used at the end of the string.
- The question mark character ('?') matches exactly one character or byte

Examples:

| | |
|---|---|
| `"Error:*"` | Any string that starts with “Error:” will match |
| `"V3.? Loaded"` | Accepts any string that says that one of the subversion of V3 is loaded |

### Assigning (part of) the string to a variable

Variable names surrounded by '#' serve as a placeholder for numeric values. The corresponding value in the received string is assigned to the variable.
This mechanism is similar to the variable substitution into strings but now from string into variable:

```
#<name>:<format>#
```

<format> is one of:

| **<format>** | |
|---|---|
| d | Decimal. Default format if no <format> specifier is present. |
| f | Floating point. The value is multiplied by 1000 and represented as an integer |
| x | Hexadecimal |
| b | Binary |
| r<n><e> | 'Raw' format. The next *n* received bytes will be NOT be considered as ASCII characters representing a value but as 'raw' byte values.<br>n: Number of bytes that must be used for the value. Default value is 1.<br>e: 'Endianess': 'l' stands for LSByte first, 'm' stands for MSByte first Default value is MSByte first. |
| s | String. The current implementation allows for one string-type variable to which all subsequent characters are assigned |

Note that there is no <size> specifier as in printing strings.

The `:<format>` part is optional. If not specified, a decimal format is assumed.

Examples:

| | |
|---|---|
| `"Weight = #W# kg"` | The variable #W will become 125 if “Weight = 125 kg” is received. |
| `":#Received:r4#.123"` | The input string matches the expected string if it starts with a ':', followed by four bytes of any value followed by “.123”. The four bytes following the ':' will be assigned to the variable #Received (MSByte first).<br>If `":\x40\x00\x01\x10.123"` is received then #Received becomes 0x40000110<br>(see 2.4.3 below for an explanation of the '\xnn' notation) |

**Note, if a literal '*', '?', '#' or '\' character is expected, then these characters shall be preceded by a backslash '\' character.**

## Special Characters in Strings: Escape Codes

Besides the normal ('printable') characters, it is possible to include non-printing characters into a string. The generic way is to specify the hex value of the character with the construction "\x<nn>" where <nn> is the two digit hexadecimal value to be inserted. For some frequently used characters, shortcuts are available, like "\n" for a newline character. See the table below.

Another use for the backslash character is to remove the special meaning from the character that follows the backslash. Examples are:

- \" to indicate that we want a double quote in the string (normally, a " would mean the end of the string)
- \\ to print a simple \ character (otherwise it is considered as the escape character)

| ***Character sequence*** | ***Effect on string*** |
|---|---|
| \# | Literal # character instead of a variable delimiter |
| \t | Tab character |
| \n | Newline character |
| \r | Return character |
| \f | Line feed character |
| \xnn | Hexadecimal value <nn><br>This format can only be used in TRANSMIT_xxx and RECEIVE_xxx commands and not in other strings like e.g. error messages and LOG messages. |
| \" | Literal quote |
| \\ | Backslash |

## Serial Communication with the UUT

Many UUTs will comprise a microcontroller that may use a serial communications link. The TEST-OK system supports serial links and allows the user to send packets to this link and receive packets from it in more than one mode.
Read the separate chapter on Serial Communication and packet definitions for all the details.

# Examples

This chapter illustrates some complete scripts.
Note that all commands are independent of the Test Controller Card (TCC) that is controlled. However, a specific TCC may not support certain functions and thus some commands will not work for that specific TCC.

## Testing analog inputs by applying a voltage and asking the result via a serial channel

In this example (with mappings on a TEST-Mate type 1), an analog input of a UUT with a microcontroller is tested. After defining mappings for the used I/O, an analog output is set to a specific voltage. Using the microprocessor's UART channel, a command is sent to the microprocessor. The microprocessor will answer with the value it 'sees' in its input.
In case there is some circuitry between the input connector of the UUT and the input pin of the microprocessor, it is useful to check whether the reported voltage on the input is really different from the expected value (because of some missing components for example) or whether the microprocessor has e.g. an incorrect reference voltage. For this purpose, a test point placed directly on the input allows to read back the actual voltage. This point is connected to an analog input of the tester and allows a check whether it is the processor that is wrong or the circuitry around it.
Below, the contents of preamble and actual test script and some explanations are shown:

### Preamble

```
VAR #SerialChannel;
VAR #Supply;

MAP $ActualIn ON ANALOG IN  101;    // 'feedback' on what is the actual voltage on the
uP input
MAP $InputPin ON ANALOG OUT 101;    // Tester output that drives the UUT input

#SerialChannel = 101;
#Supply = 101;

CONFIG_SERIAL [ #SerialChannel ] BAUDRATE = 19200, PARITY = NONE, TXMODE = MANUAL,
RXMODE = MANUAL, RXTHRESHOLD = 100, RXTIMEOUT = 10, STXMODE = OFF, EOL = CRLF;

//--- The first 500ms after power up, we allow 'unlimited' current instead of the
expected max.
//--- value of 100mA. Due to some capacitors on the UUT, there is an inrush current
that would
//--- otherwise abort the test due to over-current protection of the power supply.
CONFIG_SUPPLY [ #Supply ] VOLTAGE = 7.0, CURRENTLIMIT = 0;
SET_SUPPLY [ #Supply ] = ON;
WAITMS 500;
CONFIG_SUPPLY [ #Supply ] VOLTAGE = 7.0, CURRENTLIMIT = 0.1;
```

## Test Script

A For-loop is used to repeat the test for three different voltages applied to the input. As is explained in the section of the FOR command, the loop variable (#Applied in our example) does not need to be explicitly defined with 'VAR'.

```
VAR #Measured;
VAR #Reported;
VAR #Delta;

for #Applied { 0, 10.0, 24.0 }

    //--- Apply 0, 10 or 24V to the input connector of the UUT and wait some time to
settle
    SET_ANALOG[$InputPin] = #Applied;
    WAITMS 100;

    //--- Sending “GA3” will result in the processor measure the voltage on input 3
    //--- It will then send back a decimal number indication the voltage in mV,
    //--- e.g. “10000” for 10.000V. The '#Reported#' in the expected receive string
will convert
    //--- the received string of digits into a decimal value that is assigned to the
    //--- variable #Reported.
    TRANSMIT_SERIAL [ #SerialChannel ] "GA3";       // 'Get Analog input 3'
    RECEIVE_SERIAL [ #SerialChannel ] "#Reported#", TIMEOUT = 2000;

    //--- Measure the actual value on the processor input and store it in #Measured
    #Measured = TEST_ANALOG[$ActualIn];

    LOG "Applied: #Applied# mV, On uP input: #Measured #, reported: #Reported#",
INDENT = 4;

    //--- Calculate the difference between reported and actual voltage and make the
test fail if
    //--- the difference is more than 500mV (the built-in 'abs' function will
transform negative
    //--- differences (i.e. #Measured is bigger then #Reported) to a positive value.
    #Delta = #Reported - #Measured;
    IF ( abs(#Delta) > 500 )
        FAIL "Analog input 3 deviates too much (#Delta# mV @ #Measured# mV)";
    ENDIF;
endfor;
```

## Using arrays to repeat the same test sequence for multiple outputs

This example shows how arrays can be used to transmit different commands to a UUT with a microprocessor using a For-loop. The loop iterator (#i) is used as an index in four arrays that contain commands to be sent to the UUT (#OnCommand and #OffCommand), a text to be shown in the user dialog with the ASK command (#Question) and a failure message (#FailMsg).

Also not the use of the variable #LogMode. This allows to enable/disable logging of the TRANSMIT_SERIAL and RECEIVE_SERIAL commands by simply changing the initial value (in the line with 'VAR #LogMode = 0') instead of modifying the LOG parameter of the individual XXX_SERIAL commands. Typically it will be '1' during debugging and '0' when production testing.

```
//--------------------------------------------------------------------------------
// Test of the four LEDs on the demo board.
// Using the serial interface with the processor on the demo board, commands are sent
// that will light up the four LEDs one at a time.
// The user is asked to confirm the correct functioning of each LED
//
// (c) TEST-OK B.V. 2013
//--------------------------------------------------------------------------------

VAR # SerialChannel = 301;   // UART on TEST-Mate Type 3
VAR #LogMode = 0;
VAR #Reply;

VAR #OnCommand[]  = { "SL1 G 1", "SL1 R 1", "SL2 G 1",  "SL2 R 1" };
VAR #OffCommand[] = { "SL1 G 0", "SL1 R 0", "SL2 G 0",  "SL2 R 0" };
VAR #Question[]   = { "Is LED 1 green?", "Is LED 1 red?",  "Is LED 2 green?", "Is LED
2 red?" };
VAR #FailMsg[]    = { "Green color of LED 1 does not work", "Red color of LED 1 does
not work", "Green color of LED 2 does not work", "Red color of LED 2 does not work" };

for #i { 0 .. 3 }
    TRANSMIT_SERIAL [ #SerialChannel ] #OnCommand[#i], LOG = #LogMode;
    RECEIVE_SERIAL [ #SerialChannel ] "OK", TIMEOUT = 2000, LOG = #LogMode;
    #Reply = ASK #Question[#i], TYPE = YESNO;
    if ( #Reply == NO )
        FAIL #FailMsg[#i];  // the test fails when the user clicks 'NO', i.e. the LED
is not on endif;

    TRANSMIT_SERIAL [ #SerialChannel ] #OffCommand[#i], LOG = #LogMode;
    RECEIVE_SERIAL [ #SerialChannel ] "OK", TIMEOUT = 2000, LOG = #LogMode;
endfor;
```

## Calibrating a frequency

The CALIBRATE command is a very powerful command that can be used to calibrate any value on the Test Controller Card (TCC) inputs.
Here, an example is shown that uses the pulse width mode of one of the counter inputs of the TCC which is useful if the specific TCC does directly support a frequency measurement mode.

Two versions of the CALIBRATE command exist. Both versions require three types of information: the value to be calibrated, the range of valid values and a caption that is displayed during calibration. A dialog box will be shown with the caption, the range and the actual value while the background of the value will be red or green depending on whether the actual value is within the range or not.
The first version of the calibrate command works for analog inputs and directly uses the actual value on the specified input. The second version works on a variable which is continuously updated by a list of commands in the 'DO ... END' part of the CALIBRATE command. In this example, the second version is used.
The command fails (and thus the test) if the user clicks on the Cancel button or if the user clicks on the OK button while the frequency is outside the valid range.

The theory is simple: measure both the duration of positive and negative pulse widths and calculate the frequency by taking the reciprocal of their sum:

F = 1 / (Tpos + Tneg)

In the script below, the calculated value is assigned to variable *#Frequency* which is specified in the CALIBRATE command as the value to be calibrated.
The specified valid range is 1000..1100 (Hz).
The commands between the 'DO' and 'END' are continuously executed and form the actual measurement.
A dialog on the screen will be shown with the actual value of *#Frequency* and the specified range.
After the command is finished, the last measured frequency is available in the built-in #_IN_ variable. At the end of the script, this is used to store the value in the database as part of the test results by using a LOG command with parameter 'RECORD = ON'.

```
//-------------------------------------------------------------------------------
//--- This script measures the frequency on counter input 2 by using the
//--- pulse width mode.
//--- Note that counter 2 cannot be configured directly for frequency measurement
//--- on the TCC1800.
//---
//--- The results are accurate for frequencies between 40 Hz and 50kHz (TCC1800).
//-------------------------------------------------------------------------------
Var #Pos;
Var #Neg;
Var #Frequency;

Calibrate VAR [#Frequency] { 1000 .. 1100 }, Caption = "Frequency on input 2", Unit =
"Hz", Decimal  = 0 DO

    Config_Counter [2] Mode = PULSEWIDTH, Edge = POS;

    //--- Wait until the width of the positive part of the wave is measured.
    //--- On timeout, do not generate a 'fail'
    WaitWhile (1000) Else ignore, "No signal present on input 2"
        Test_Counter[2] Expect == 0;

    #Pos = Test_Counter[2];    // pulse width in us

    Config_Counter [2] Mode = PULSEWIDTH, Edge = NEG;

    //--- Wait until the width of the negative part of the wave is measured.
    WaitWhile (1000) Else ignore, "No signal present on input 2"
        Test_Counter[2] Expect == 0;
```

```
    #Neg = Test_Counter[2];

    If ( #Neg == 0 or #Pos == 0 )
        //--- show '0' if no signal is present
        #Frequency = 0;
    Else
        #Frequency = 1000000 / (#Pos + #Neg);
    Endif;
End;

//--- Store the measured frequency with the test results in the database
Log “F = #_IN_#Hz”, RECORD = ON;
```

# Detailed Command Description

This chapter is a reference for all commands that are supported in the TEST-OK test language.

Before going in the details of each command, it is important to explain something about the parameter values that represent a voltage or a current:

- **when the number contains a decimal point, then the value is in V or A.**
- **when an integer number is used (i.e. without a decimal point) then the value is either in mV or mA.**

Floating point (constant) values are always converted to integer values by multiplying them with 1000. The historical reason for this is that commands like TEST_ANALOG and SET_ANALOG work with values in mV so you must e.g. specify *SET_ANALOG[1] = 3300* to set the output to 3.3V. In order to allow for the more comprehensive notation of *SET_ANALOG[1] = 3.3* floating point values are silently converted to integer via multiplication of 1000.
This also implies that only the first 3 digits after the decimal point will 'survive' in the conversion: 1.2345 will be multiplied by 1000 and converted to integer resulting in 1234.

Examples:

```
// Set analog output 1 to 125 mV
SET_ANALOG [1] = 125;

// Set  analog output 3 to 1 V (1000 mV)
SET_ANALOG [3] = 1.0;   =>

// Set power supply 301 to 5V with a maximum current of 200 mA
CONFIG_SUPPLY[301] VOLTAGE =  5.000, CURRENTLIMIT = 200;

TEST_ANALOG [ $V_opamp ] EXPECT ((>6)and(<10)), "Voltage out of range";
```

## SET_SUPPLY

**SET_SUPPLY [ *Channel* ] = *Value*;**

**Description:**

This command activates / deactivates a power supply. Which power supply is controlled is specified by *Channel* which can be either a constant or a variable.

The built-in variable #_OUT_ is set to the assigned *Value*.

**Parameters:**

| *Name* | *Value* | *Unit* | *Description* |
|---|---|---|---|
| Channel | 1...n | - | Indicates the specific power supply that must be switched on or off. |
| Value | ON, OFF or Variable | - | Indicates whether the power supply must be switched ON or OFF. Any value other then 0 will switch on the supply |

**Example:**

```
SET_SUPPLY [ 1 ] = ON;
```

To set multiple power supplies on:

```
VAR #i;

FOR #i { 1 .. 3 }
    SET_SUPPLY [ #i ] = ON;
ENDFOR;
```

## SET_DIGITAL

**SET_DIGITAL [ GROUP *GroupNumber*, BIT *BitNumber* ] = *Value*;**
**SET_DIGITAL [ GROUP *GroupNumber*, BIT *FirstBit .. LastBit* ] = *Value*;**
**SET_DIGITAL [ *$MapName* ] = *Value*;**

**Description**:

Sets/resets one or more digital output(s) of the same group. Which group and which bit(s) is either specified by the GROUP and BIT parameters or by specifying a mapping name (see the MAP command in section 4.4.1).
The lowest bit of the bits to be set will be equal to the least significant bit of the specified Value (e.g. setting output bits 3..5 of a specific group to 0b100 will set bits 3 and 4 to 0 and bit 5 to 1)

The built-in variable #_OUT_ is set to the assigned *Value*.

Note that 'setting' an output means activating the output. If the corresponding output is an open drain output with a pull-up resistor connected to it, then setting the output 'ON' will result in a logical '0' in the physical output.

**Parameters**:

| Name | Value | Unit | Description |
|---|---|---|---|
| GroupNumber | 1...n | - | Specifies the group in which a bit must be set |
| BitNumber, FirstBit, LastBit | 1...n | - | The lowest bit in a group is bit number 1. There is no bit number 0! |
| Value | ON, OFF, 0...n or a variable. | - | ON will result in all specified bits to be set while OFF results in all specified bits to be reset.<br>If an integer value is specified then the lower <x> bits of the specified value will be used to set the output bits where <x> is the number of bits to be set. |
| MapName | String starting with $ | - | Substitute for group and bit number(s) as defined with a MAP command. |

**Examples:**

The next three lines do the following:

1) sets bit 1 of group 2,
2) sets bit 3, 4, and 5 of group 5
3) sets bit 2 and 4 and resets bit 1 and 3 of group 5

```
SET_DIGITAL [ GROUP 2, BIT 1 ] = ON;
SET_DIGITAL [ GROUP 5, BIT 3 .. 5 ] = ON;
SET_DIGITAL [ GROUP 5, BIT 1 .. 4 ] = 0b1010;
```

The next two lines use a MAP command to associate a name to a group of bits. This name is then used to set all the bits in a single command. Bits 5 and 8 will be set to 1 and the other bits will be set to 0.

```
MAP $Address ON DIGITAL OUT GROUP 1, BIT 1..8;
SET_DIGITAL [  $Address ] = 0b10010000;
```

Note that most of the time the MAP command will be defined in the Preamble script so that it can be used

in several tests without having to define it multiple times.

The next example shows the use of a variable to determine what shall be set on the outputs (group 202 refers to a TEST-Mate)

```
MAP $ControlCode ON DIGITAL OUT GROUP 202 BIT 1..4

FOR #Control { 0x2, 0x5, 0x7 }
    SET_DIGITAL [  $ControlCode ] = #Control;
    LOG "Setting #Control# on the Control bits", INDENT = 2;

    WAITMS 500;   // give some time to the UUT to process the new code

    .....  // test the required response of the UUT
ENDIF
```

## SET_ANALOG

**SET_ANALOG [ *Channel* ] = *Value*;**

**Description:**

Sets/resets the specified analog output.

**SET_ANALOG [ *$MapName* ] = *Value*;**

**Description:**

Sets/resets the group of analog outputs defined by the variable.

The built-in variable #_OUT_ is set to the assigned *Value*.

**Parameters:**

| *Name* | *Value* | *Unit* | *Description* |
|---|---|---|---|
| Channel | 1...n | - | Specifies the analog output channel to be set |
| Value | Integer or Float | mV or V | The value expresses a voltage. A constant value may either contain a decimal point or not. If there is a decimal point, then the value is in volts. Otherwise the value is in mV. When using a variable, then the value is always in mV. |
| MapName | String starting with $ | - | Defined with Map command. |

**Example:**

```
// Because there is a decimal point, the value is in volts( i.e. 2000 mV )
SET_ANALOG [ 1 ]    = 2.000;

// No decimal point, therefore the value is in mV
SET_ANALOG [ 10 ]   = 2;

// The map name $Ntc shall be declared prior to this line
SET_ANALOG [ $Ntc ] = 3.345;

// The result of these two lines is the same as the example above
VAR #Value = 3345;
SET_ANALOG [ $Ntc ] = #Value;
```

## SET_PWM*

**SET_PWM [ *Channel* ] = Value;**

*Not available in My-TEST and eC-my-test

**Description:**

Activates / de-activates the specified PWM channel.

The built-in variable #_OUT_ is set to the assigned *Value*.

See also: CONFIG_PWM

**Parameters:**

| *Name* | *Value* | *Unit* | *Description* |
|---|---|---|---|
| Channel | 1...n | - | Indicates the specific PWM channel that must be switched on or off. |
| Value | ON, OFF or Variable | - | Indicates whether the PWM channel must be switched ON or OFF. Any value other then 0 will switch on the PWM |

**Example:**

```
//--- Activate PWM 1
SET_PWM [ 1 ]  = ON;

//--- PWM 2 is active (3 kHz, 50% duty cycle) if digital input 10 is high
VAR #OnOff;

CONFIG_PWM [2] Frequency = 3000, duty = 50;

#OnOff = TEST_DIGITAL[ 10 ];
SET_PWM [ 2 ] = #OnOff;
```

## SET_USB*

**SET_PWM [ *Port* ] = *Value*;**

*Not available in My-TEST and eC-my-test

**Description:**

Activates / de-activates the specified USB port. See the Hardware Reference Manual of the Test Controller Card for port numbers (if supported).

The built-in variable #_OUT_ is set to the assigned *Value*.

**Parameters:**

| Name | Value | Unit | Description |
|---|---|---|---|
| Port | 1...n | - | Indicates the specific USB port that must be switched on or off. |
| Value | ON, OFF or Variable | - | Indicates whether the USB port must be switched ON or OFF. Any value other then 0 will switch on the port. |

**Example:**

```
//--- Activate USB port 1
SET_USB [ 1 ]  = ON;

//--- USB 2 is active if digital input 10 is high
VAR #OnOff;
#OnOff = TEST_DIGITAL[ 10 ];
SET_USB [ 2 ] = #OnOff;
```

## PROGRAM*

**PROGRAM [ *Channel* ] = "*Hex File*", TYPE = <*Type*>;**
**PROGRAM [ *Channel* ] = "*Hex File*";**
**PROGRAM [ *Channel* ];**

*Not available in My-TEST and eC-my-test

**Description:**

Uses the specified programmer to program a hex file into the UUT. The first form described above specifies the complete path to a file in Intel Hex format. The file will be read and programmed into the UUT's microcontroller.
The second form does not specify a Hex file and can only be used if the application that uses the TDL language supports the selection of the hex file prior to test execution. This is for instance the case for Test-Track in which the hex file to be programmed is determined by settings in the database (as part of an order line).

The optional TYPE parameter can be used to indicate whether the specified hex file is a production hexfile or a test hex file.
Specifying PRODUCTION type means that programming is done with production software and the hex file specified by the command may be overruled by another hex file by the application. In Test-Track the specified file will be overruled by the hex file specified in the order line to which the UUT belongs.

When TEST is chosen as the TYPE, then the hex file specified in the command will not be overruled by the application. This is mainly meant to load a special software version into the UUT that is used for testing. E.g. the last test script in a test suite may then replace this by an official production version dictated by the application.

It is also possible to assign the 'result' of the command to a variable. The result is either 1 if programming succeeded or 0 when a problem is encountered during programming. Testing on the result allows the appropriate actions to be taken. It is e.g. useful to cancel all following tests when the processor on the UUT could not be programmed (see also the example below).

**The hex file shall contain all information that must be programmed into the processor. This includes configuration words (mandatory) and optionally EEPROM contents**

Note that it is *necessary* to configure the programmer with the CONFIG_PROGRAMMER command before using PROGRAM.

Encrypted HEX files
It is possible to read encrypted hex files that were previously encrypted with the TEST-CRYPT utility. Such files will be recognized by the '.hexx' file name extension. If such a file is specified, then this file will be decoded using the serial number of the dongle that is currently connected to the PC.

**Parameters:**

| Name | Value | Unit | Description |
|---|---|---|---|
| Channel | 1...n | - | Indicates the programmer channel that must perform the programming. |
| Hex File (Optional) | String | - | HEX file |
| Type (Optional) | **PRODUCTION**, TEST | - | Type of hex file. A PROD file means that this is production software and that the actual file used may be overwritten by a |

| Name | Value | Unit | Description |
|---|---|---|---|
| | | | hex file determined by the application (in case of Test-Track, the hex file specified in the order line on which the board occurs).<br>A TEST file means that this is software used for the tests only and it will never be replaced like a PROD type file can. |

**Examples:**

```
CONFIG_PROGRAMMER[1]  "PIC18F2420";

PROGRAM [1] = "D:/Hex Files/MFDC8-1RE526I.hex";
```

A more elaborate example used in the TEST-TRACK application retrieves the hex file from the database and will not continue subsequent tests when programming fails:

```
VAR #Result;

CONFIG_PROGRAMMER[1]  "PIC16C620A";

#Result = PROGRAM [1];

IF ( #Result != 1 )
    FAIL "Programming failed. No use to continue", ABORT_ALL;
ENDIF;
```

## RESET_COUNTER*

| **RESET_COUNTER [** ***Channel*** ]; |
|---|

*Not available in My-TEST and eC-my-test

**Description:**

Resets the specified counter to 0.

**Parameters:**

| ***Name*** | ***Value*** | ***Unit*** | ***Description*** |
|---|---|---|---|
| Channel | 1...n | - | Indicates the counter channel that must be reset |

**Example:**

See CONFIG_COUNTER command

## SET_SERIALFIELD*

**SET_SERIALFIELD [ *Channel* ] $*FieldName* = *Value*;**

*Not available in My-TEST and eC-my-test

**Description:**

Modifies the specified field of the packet that is sent over the specified serial channel to the defined value.

**Parameters**:

| ***Name*** | ***Value*** | ***Unit*** | ***Description*** |
|---|---|---|---|
| Channel | 1...n | - | |
| $FieldName | String starting with $ | - | |
| Value | 0 .. n | - | Integer or string value.<br>Integer value: if the field is n bytes wide, then the least significant n bytes of the specified value will be used to set the field.<br>String value: if the field is n bytes wide, then up to n characters from the string are copied tot the field. If the string has less characters then there are bytes in the field, then the remaining field bytes will be set to 0. |

**Example:**

```
SET_SERIALFIELD [ 1 ] $ID = 15;
SET_SERIALFIELD [ 1 ] $ID = #Value;
SET_SERIALFIELD [ 1 ] $SerialNum = #_SERIAL_NUMBER_;
SET_SERIALFIELD [ 1 ] $Cmd = "Get IN1";
```

## TRANSMIT_SERIAL

**TRANSMIT_SERIAL [ *Channel* ] ;**
**TRANSMIT_SERIAL [ *Channel* ] "*Message*" ;**
**TRANSMIT_SERIAL [ *Channel* ] , LOG = *OnOff*;**
**TRANSMIT_SERIAL [ *Channel* ] "*Message*", LOG = *OnOff* ;**

**Description**:

When the serial channel is configured in 'manual' mode, this command will send a packet to the UUT.

The first form listed above does not specify a message to be sent and it can only be used if a Packet Definition file is specified. When executed, it will transmit the packet as defined in the Packet Definition file with the current field values.

The second form can only be used when the communication is in Terminal Mode (i.e. no Packet Definition file is specified) and it will transmit the specified ASCII string. The command will automatically add a CR and or LF character at the end of the string (unless configured otherwise with CONFIG_SERIAL).
Furthermore, variable substitution as described in section 2.4.1 is also supported which makes it possible to send packets with variable content in e.g. a FOR-loop (see the examples below).

Terminal mode is characterized by a question-answer mechanism that is lead by the tester (i.e. the script commands). The UUT is supposed to send one or more strings (answers) in response to a string sent by the tester (question). In order to synchronize the answers, the TRANSMIT_SERIAL command will flush the receive buffer so that following RECEIVE_SERIAL commands are guaranteed to process data that was sent after the last 'question'.
This mode is exactly the same as used for the TRANSMIT_COM and TRANSMIT_TCP commands. Examples for these commands may be used as well. See section 6.2 for more information.

Also see the description of the CONFIG_SERIAL command and the chapter on Serial Communication.

**Parameters**:

| ***Name*** | ***Value*** | ***Unit*** | ***Description*** |
|---|---|---|---|
| Channel | 1...n | - | Specifies the serial channel to use |
| Message | String | - | Used in Terminal mode only, it is the ASCII string that is to be sent to the UUT. |
| Log (optional) | ON, **OFF** or 0...n | - | Any integer value other than 0 will be interpreted as 'ON' Value '2' has a special meaning: all received bytes are printed in hex format, even if the byte as a printable (ASCII) value. |

**Example:**

In Single Packet Mode, set a field and send a predefined packet on channel 1:

```
SET_SERIALFIELD [1] $SELFTEST = 1;
TRANSMIT_SERIAL [1];
```

In Terminal Mode, activate different outputs by sending ASCII commands on channel 2:

```
FOR #i { 1 ..4 }
    TRANSMIT_SERIAL [2] "SET OUTPUT#i# = ON";
ENDFOR;
```

## TRANSMIT_CAN

**TRANSMIT_CAN [ *Channel* ] "*Message*", ID = *Id* ;**
**TRANSMIT_CAN [ *Channel* ] "*Message*", ID = *Id* , LOG = *OnOff* ;**

**Description**:

This command will send a packet to the specified CAN interface on the Test Controller Card.

The command will transmit the specified packet using the specified ID. Whether a standard or extended ID is transmitted can be set with the CONFIG_CAN command. Variable substitution as described is supported which makes it possible to send packets with variable content in e.g. a FOR-loop (see the examples below).

Communication over the CAN channel is characterized by a question-answer mechanism that is lead by the tester (i.e. the script commands). The UUT is supposed to send one or more packets (answers) in response to a packet sent by the tester (question). In order to synchronize the answers, the TRANSMIT_CAN command will flush the receive buffer so that following RECEIVE_CAN commands are guaranteed to process data that was sent after the last 'question'.
This mode is exactly the same as used for the TRANSMIT_COM and TRANSMIT_TCP commands. Examples for these commands may be used as well. See Terminal Mode for more information.

The built-in variable #_CAN_STATUS_ (which contain error flags for errors that occur on the CAN bus) is cleared just before the transmission of the packet. More details of the #_CAN_STATUS_ can be found in the section of the RECEIVE_CAN command.

Also see the description of the CONFIG_CAN command and the chapter on Serial Communication and packet definition.

**Parameters**:

| ***Name*** | ***Value*** | ***Unit*** | ***Description*** |
|---|---|---|---|
| Channel | 1...n | - | Indicates which CAN channel (device) on the Test Controller Card is concerned. |
| Message | String | - | Data that is to be sent to the UUT. |
| Id | 1..n | - | ID of the CAN packet. Whether a standard or extended ID is transmitted can be set with the CONFIG_CAN command. |
| Log (optional) | ON, **OFF** or 0...n | - | Any integer value other than 0 will be interpreted as 'ON' Value '2' has a special meaning: all received bytes are printed in hex format, even if the byte as a printable (ASCII) value. |

**Example:** see CONFIG_CAN and RECEIVE_CAN

## TRANSMIT_RS485

**TRANSMIT_RS485 [ *Channel* ] "*Message*" ;**
**TRANSMIT_RS485 [ *Channel* ] "*Message*", LOG = *OnOff* ;**

**Description**:

This command will send a packet to the RS485 interface on the Test Controller Card.

The command will transmit the specified packet. The command will automatically add a CR and or LF character at the end of the string (unless configured otherwise with CONFIG_RS485).
Furthermore, variable substitution as described in section Values in output strings is also supported which makes it possible to send packets with variable content in e.g. a FOR-loop (see the examples below).

Communication over the RS485 channel is characterized by a question-answer mechanism that is lead by the tester (i.e. the script commands). The UUT is supposed to send one or more strings (answers) in response to a string sent by the tester (question). In order to synchronize the answers, the TRANSMIT_RS485 command will flush the receive buffer so that following RECEIVE_RS485 commands are guaranteed to process data that was sent after the last 'question'.
This mode is exactly the same as used for the TRANSMIT_COM and TRANSMIT_TCP commands. Examples for these commands may be used as well. See section Terminal Mode for more information.

Also see the description of the CONFIG_RS485 command and the chapter on Serial Communications and packet definitions.

**Parameters**:

| ***Name*** | ***Value*** | ***Unit*** | ***Description*** |
|---|---|---|---|
| Channel | 1...n | - | Specifies the serial channel to use |
| Message | String | - | Data that is to be sent to the UUT. |
| Log (optional) | ON, **OFF** or 0...n | - | Any integer value other than 0 will be interpreted as 'ON' Value '2' has a special meaning: all received bytes are printed in hex format, even if the byte as a printable (ASCII) value. |

## TRANSMIT_COM*

**TRANSMIT_COM [ *Channel* ] "*Message*" ;**
**TRANSMIT_ COM [ *Channel* ] "*Message*", LOG = *OnOff* ;**

*Not available in My-TEST and eC-my-test

**Description**:

This command will transmit the specified ASCII string to the PC's COM port that was configured with the CONFIG_COM command.

The command will automatically add a CR and or LF character at the end of the string (unless configured otherwise with CONFIG_COM).
Furthermore, variable substitution as described in section Values in output strings is also supported which makes it possible to send packets with variable content in e.g. a FOR-loop (see the examples below) and even binary (non-ASCII) packets (see section Special characters in strings: escape codes). Also see section Serial Communications and packet definitions for more information.

The communication over the COM port is characterized by a question-answer mechanism that is lead by the tester (i.e. the script commands). The external device (e.g an oscilloscope) is supposed to send one or more strings (answers) in response to a string sent by the tester (question). In order to synchronize the answers, the TRANSMIT_COM command will flush the receive buffer so that following RECEIVE_COM commands are guaranteed to process data that was sent after the last 'question'.

Also see the description of the CONFIG_COM command

**Parameters**:

| ***Name*** | ***Value*** | ***Unit*** | ***Description*** |
|---|---|---|---|
| Channel | 1...n | - | Specifies the channel from which data shall be received. See the CONFIG_COM command. |
| Message | String | - | The ASCII string that is to be sent to the COM port. |
| OnOff (optional) | ON, **OFF** or 0...n | - | Any integer value other than 0 will be interpreted as 'ON'. Value '2' has a special meaning: all received bytes are printed in hex format, even if the byte as a printable (ASCII) value. |

**Examples:**

Initialize a Tektronix TDS2014 Oscilloscope that is connected on COM14 to 50V/div and 50us/div:

```
VAR #COM = 14;

CONFIG_COM [ #COM ] BAUDRATE = 19200, PARITY = NONE, EOL = LF;

TRANSMIT_COM [ #COM ] ":SELECT:CH1 ON", LOG = ON;
TRANSMIT_COM [ #COM ] ":SELECT:CH2 OFF", LOG = ON;
TRANSMIT_COM [ #COM ] ":SELECT:CH3 OFF", LOG = ON;
TRANSMIT_COM [ #COM ] ":SELECT:CH4 OFF", LOG = ON;

TRANSMIT_COM [ #COM ] ":CH1:coupling DC", LOG = ON;
TRANSMIT_COM [ #COM ] ":CH1:scale 50", LOG = ON;      // 50V/div

TRANSMIT_COM [ #COM ] ":HOR:delay:scale 50E-6", LOG = ON;
```

'LOG = ON' means the commands are echoed on the log screen. This may not be necessary. Simply remove the parameter or set it to 'OFF'.
The above example uses a variable #COM to set the COM port. This allows easy changing of the COM port without having to change all the individual COM commands.

It is also possible to send non-ASCII bytes by using the escape code mechanism (explained in section Special characters in strings: escape codes).
Sending the byte sequence 0x0F 0x0F 0x3A 0x22 0x05 over comport COM5 can be done in the following way (note 'EOL = NONE' in the configuration: no additional CR and/of LF characters are sent):

```
CONFIG_COM [ 5 ] BAUDRATE = 19200, PARITY = NONE, EOL = NONE;

TRANSMIT_COM [ 5 ] "\x0f\x0f\x3a\x22\x05", LOG = ON;
```

## TRANSMIT_TCP*

**TRANSMIT_TCP [ *Channel* ] "*Message*" ;**
**TRANSMIT_TCP [ *Channel* ] "*Message*", LOG = *OnOff* ;**

*Not available in My-TEST and eC-my-test

**Description**:

This command will transmit the specified ASCII string to the external application that is listening on the TCP port that was configured with the CONFIG_TCP command.

The command will automatically add a CR and or LF character at the end of the string (unless configured otherwise with CONFIG_TCP).
Furthermore, variable substitution as described in section Values in output strings is also supported which makes it possible to send packets with variable content in e.g. a FOR-loop (see the examples below) and even binary (non-ASCII) packets (see section Special characters in strings: escape codes).
Also see section Terminal Mode for more information.

The communication with the TCP server application is characterized by a question-answer mechanism that is lead by the tester (i.e. the script commands). The external application is supposed to send one or more strings (answers) in response to a string sent by the tester (question). In order to synchronize the answers, the TRANSMIT_TCP command will flush the receive buffer so that following RECEIVE_TCP commands are guaranteed to process data that was sent after the last 'question'.

Also see the description of the CONFIG_TCP command

**Parameters**:

| Name | Value | Unit | Description |
|---|---|---|---|
| Channel | 1...n | - | Specifies the channel from which data shall be received. See the CONFIG_TCP command. |
| Message | String | - | The ASCII string that is to be sent to the TCP server. |
| OnOff (optional) | ON, **OFF** or 0...n | - | Any integer value other than 0 will be interpreted as 'ON'. Value '2' has a special meaning: all received bytes are printed in hex format, even if the byte as a printable (ASCII) value. |

**Examples:**

See also the RECEIVE_TCP* command.

## TRANSMIT_VISA*

**TRANSMIT_TCP [ *Channel* ] "*Message*" ;**
**TRANSMIT_TCP [ *Channel* ] "*Message*", LOG = *OnOff* ;**

*Not available in My-TEST and eC-my-test

**Description**:

This command will transmit the specified ASCII string to the (test-)instrument that was configured with the CONFIG_VISA command.

The command will automatically add a LF character at the end of the string.
Furthermore, variable substitution as described in section Values in output strings is also supported which makes it possible to send packets with variable content in e.g. a FOR-loop (see the examples below) and even binary (non-ASCII) packets (see section Special characters in strings: escape codes).
Also see section Terminal Mode for more information.

Also see the description of the CONFIG_VISA command

**Parameters**:

| *Name* | *Value* | *Unit* | *Description* |
|---|---|---|---|
| Channel | 1...n | - | Specifies the channel from which data shall be received. See the CONFIG_VISA command. |
| Message | String | - | The ASCII string that is to be sent to the VISA instrument. |
| OnOff (optional) | ON, **OFF** or 0...n | - | Any integer value other than 0 will be interpreted as 'ON'. Value '2' has a special meaning: all received bytes are printed in hex format, even if the byte as a printable (ASCII) value. |

**Examples**:

See also the RECEIVE_VISA* command.

## I2C_START

| **I2C_START [ *Channel* ]** |
|---|

**Description**:

This command generates a START condition on the specified I²C interface. A START condition is defined as changing SDA from high to low while SCL is high.
SCL is explicitly set to high before the SDA transition occurs so that the command also works as a re-START which is sometimes required during a transaction (see example)

**Parameters**:

| ***Name*** | ***Value*** | ***Unit*** | ***Description*** |
|---|---|---|---|
| Channel | 1...n | - | Specifies the I²C channel to use |

**Examples:**

Write a byte to a serial eeprom (a Microchip 24LC00) and read it back for verification.

```
Var #Rd;

//--- Write 0x55 to address 0x10
I2c_Start[1];
I2c_Write[1] = 0xa0;
I2c_Write[1] = 0x10;
I2c_Write[1] = 0x55;
I2c_Stop[1];

//--- Read from address 0x10 and compare with 0xaa
I2c_Start[1];
I2c_Write[1] = 0xa0, ELSE IGNORE;  // ignore the ACK bit
I2c_Write[1] = 0x10;                    // Address = 0x10
I2c_Start[1];                    // re-START
I2c_Write[1] = 0xa1;                    // Read (from address 0x10)
#Rd = I2c_Read[1];
I2c_Stop[1];

if ( #Rd != 0x55 )
    Fail "Reading wrong value from address 1: #Rd:x# instead of 0x55",
Abort;
else
    Log "OK", indent = 4;
endif;
```

## I2C_STOP

**I2C_STOP [ *Channel* ]**

**Description**:

This command generates a STOP condition on the specified I²C interface. A STOP condition is defined as changing SDA from low to high while SCL is high.

**Parameters**:

| *Name* | *Value* | *Unit* | *Description* |
|---|---|---|---|
| Channel | 1...n | - | Specifies the I²C channel to use |

**Examples:**

See I2C_START

## I2C_WRITE

**I2C_WRITE [ *Channel* ] = *Value*;**
**I2C_WRITE [ *Channel* ] = *Value ELSE ConditionMode*;**

**Description**:

This command writes the lower 8 bits of the specified *Value* to the specified I²C interface. And then reads back the ACK bit from the slave. If NACK ('1') is received, then the command will fail.

**Parameters**:

| Name | Value | Unit | Description |
|---|---|---|---|
| Channel | 1...n | - | Specifies the I²C channel to use |
| Value | 0..255 | - | Byte to be written on the I²C bus |
| ConditionMode<br>(Optional) | IGNORE,<br>**CONTINUE** ,<br>ABORT,<br>ABORT_ALL | - | If a NACK is received from the slace, then the condition mode indicates whether the test shall be aborted immediately or whether subsequent tests shall still be executed.<br>By default ConditionMode = CONTINUE so if this parameter is omitted subsequent tests will be executed.<br>If mode IGNORE is specified, then the command will never cause the test to fail. In this case, there will also not be a message on the log screen. |

**Examples:**

See I2C_START

## I2C_READ

**I2C_READ [ *Channel* ];**

**Description**:

This command reads one byte from the specified I²C interface (and acknowledges the reception). The result can be assigned to a variable for further processing.

**Parameters**:

| *Name* | *Value* | *Unit* | *Description* |
|---|---|---|---|
| Channel | 1...n | - | Specifies the I²C channel to use |

**Examples:**

See I2C_START

## Testing

All commands in this section have the same structure: command name, specifications of what to test, expected value, error message and condition mode. Error message and condition mode are optional.
The expected value is also required if the command is used 'standalone', i.e. it is used to set the test verdict to fail in case the value does not match the condition. However, when the TEST_xx command is part of a variable assignment, then the expected value need not to be specified (but it still can!).

*Note that the assignment to a variable is done after the command has been executed. Using the variable in e.g. the error message will result in possibility confusing results since the variable will still have its 'old' value and not the value measured by the TEST_xx command.*

Some commands, RECEIVE_xxx and CALIBRATE deviate from the generic form. It is thought that e.g. using the word 'receive' covers better what the command is doing.

### #_ERROR_ : Handling errors programmatically

If errors occur, an appropriate message will be displayed on the screen but it is sometimes useful to execute different commands depending on whether an error occurred or not.
Two examples may explain this:

- when the voltage on a specific input is not within the expected range, we may want to skip some test commands on other inputs since we know they will fail as well, avoiding useless error messages.
- when trying to receive data from the UUT (RECEIVE_SERIAL) a timeout may occur because the UUT is not ready yet. In this case, the script may decide to repeat some initialization and try again.

The built-in variable #_ERROR_ provides the necessary information. The table below shows the meaning of its possible values.

*Note that no detailed error codes are provided for all kind of possible failures like missing configuration, incorrect parameter values etc. These failures shall not occur in a correctly written script. If they occur, one must use the information on the log screen in order to correct the problem.*

| ***Value of #_ERROR_*** | ***Description*** |
|---|---|
| 0 | Command executed without failure |
| 1 | Timeout occurred |
| 2 | Other failure (values not within EXPECTed range, mismatch in received data etc....) |
| 3 | Test aborted by the user |

The #_ERROR_ variable is set by any of the commands in this section (Testing) and by the WAITWHILE, WAITTX and WAITRX commands. All other commands do not modify its value.

## TEST_SUPPLYCURRENT

**TEST_SUPPLYCURRENT [ *Channel* ];**
**TEST_SUPPLYCURRENT [ *Channel* ] EXPECT *Condition*;**
**TEST_SUPPLYCURRENT [ *Channel* ] EXPECT *Condition*, "*ErrorMessage*";**
**TEST_SUPPLYCURRENT [ *Channel* ] EXPECT *Condition* ELSE *ConditionMode*;**
**TEST_SUPPLYCURRENT [ *Channel* ] EXPECT *Condition* ELSE *ConditionMode*, "*ErrorMessage*";**

**Description**:

Reads the supply's current and compares the returned value with the expected value defined in the condition part.
The current is measured in mA.

**Parameters**:

| ***Name*** | ***Value*** | ***Unit*** | ***Description*** |
|---|---|---|---|
| Channel | 1...n | - | Indicates the specific power supply for which the current must be tested |
| Condition | | - | See section 5.1 on page 148 for its format. |
| ConditionMode (Optional) | IGNORE, **CONTINUE** , ABORT, ABORT_ALL | - | If the condition fails, then the condition mode indicates whether the test shall be aborted immediately or whether subsequent tests shall still be executed. By default ConditionMode = CONTINUE so if this parameter is omitted, subsequent tests will be executed. |
| ErrorMessage (Optional) | String | - | When the condition fails, the specified text string is shown on the log screen |

**Example:**

```
#Current = TEST_SUPPLYCURRENT [ 1 ];
TEST_SUPPLYCURRENT [ 1 ] EXPECT ( !=0 );
TEST_SUPPLYCURRENT [ 1 ] EXPECT ( !=0 ), "Error message 1";

TEST_SUPPLYCURRENT [ 201 ] EXPECT ( !=0 ) ELSE CONTINUE;
TEST_SUPPLYCURRENT [ 301 ] EXPECT ( !=0 ) ELSE ABORT, "Error message 2";
```

## TEST_DIGITAL

**TEST_DIGITAL [ *Bits* ];**
**TEST_DIGITAL [ *Bits* ] EXPECT *Condition*;**
**TEST_DIGITAL [ *Bits* ] EXPECT *Condition*, "*ErrorMessage*";**
**TEST_DIGITAL [ *Bits* ] EXPECT *Condition* ELSE *ConditionMode*;**
**TEST_DIGITAL [ *Bits* ] EXPECT *Condition* ELSE *ConditionMode*, "*ErrorMessage*";**

**TEST_DIGITAL [ $*MapName* ] EXPECT *Condition*;**
**TEST_DIGITAL [ $*MapName* ] EXPECT *Condition*, "*ErrorMessage*";**
**TEST_DIGITAL [ $*MapName* ] EXPECT *Condition* ELSE *ConditionMode*;**
**TEST_DIGITAL [ $*MapName* ] EXPECT *Condition* ELSE *ConditionMode*, "*ErrorMessage*";**

**Description:**

Reads the specified range of digital inputs and compares the returned value with the expected value defined in the condition part.
The format for <Bits> is either a single BitNumber indicating an individual input or a range consisting of two BitNumber separated by to dots indicating a range of inputs to be tested (see examples).

*Note: using the keyword 'ON' in the condition (e.g. '== ON' ) only works when one input bit is tested since 'ON' will be replaced by the value '1'. See the examples below.*

**Parameters:**

| ***Name*** | ***Value*** | ***Unit*** | ***Description*** |
|---|---|---|---|
| BitNumber | 1...n or a variable (string starting with #) | - | Specifies the digital input bit to be tested.<br>The numbering of the digital inputs start with number 1. |
| MapName | String | - | Defined with MAP command. |
| Condition | | - | See section 5.1 on page 148 for its format. |
| ConditionMode<br>(Optional) | IGNORE, **CONTINUE**, ABORT, ABORT_ALL | - | If the condition fails, then the condition mode indicates whether the test shall be aborted immediately or whether subsequent tests shall still be executed.<br>By default ConditionMode = CONTINUE so if this parameter is omitted subsequent tests will be executed. |
| ErrorMessage<br>(Optional) | String | - | When the condition fails, the specified text string is shown on the log screen |

**Examples:**

Tests on single input bits can use either 0 or OFF and 1 or ON.

```
TEST_DIGITAL [ 1 ] EXPECT ( == 0 );
TEST_DIGITAL [ 2 ] EXPECT ( == OFF ), "Error message 1";
TEST_DIGITAL [ 3 ] EXPECT ( == 1   ) ELSE CONTINUE;
TEST_DIGITAL [ 4 ] EXPECT ( == ON  ) ELSE ABORT, "Error message 2";
```

As noted in the command description, 'ON' can only be used when tesing single bit inputs.
The following example will fail if both inputs (3 and 4) are actually '1':

```
TEST_DIGITAL[ 3 .. 4] EXPECT == ON;          // Test will FAIL
```

The correct way to test is:

```
TEST_DIGITAL[ 3 .. 4] EXPECT == 0b11;  // Test will PASS
```

Test whether inputs 301 to 308 are all zero.
Solution 1:

```
TEST_DIGITAL [ 301 .. 308 ] EXPECT ( == OFF ), "Some inputs are not off";
```

Solution 2:

```
FOR #input { 5 .. 20 }
    TEST_DIGITAL [ #input ] EXPECT ( == OFF ), "Input #input# is not off";
ENDFOR;
```

Using the MAP command, a number of input bits get a better understandable name and can be tested as a single entity:

```
MAP $SD_OUTPUTS ON DIGITAL IN BIT 1 .. 4;
TEST_DIGITAL [ $SD_OUTPUTS ] EXPECT ( != 3 );
TEST_DIGITAL [ $SD_OUTPUTS ] EXPECT ( != 0b11 ), "Error message 1";
TEST_DIGITAL [ $SD_OUTPUTS ] EXPECT ( != 0x03 )   ELSE CONTINUE;
TEST_DIGITAL [ $SD_OUTPUTS ] EXPECT ( != 0b1001 ) ELSE ABORT, "Error
message 2";
```

## TEST_ANALOG

**TEST_ANALOG [ *Channel* ];**
**TEST_ANALOG [ *Channel* ] EXPECT *Condition*;**
**TEST_ANALOG [ *Channel* ] EXPECT *Condition*, "*ErrorMessage*";**
**TEST_ANALOG [ *Channel* ] EXPECT *Condition* ELSE *ConditionMode*;**
**TEST_ANALOG [ *Channel* ] EXPECT *Condition* ELSE *ConditionMode*, "*ErrorMessage*";**

**TEST_ANALOG [ $*MapName* ];**
**TEST_ANALOG [ $*MapName* ] EXPECT *Condition*;**
**TEST_ANALOG [ $*MapName* ] EXPECT *Condition*, "*ErrorMessage*";**
**TEST_ANALOG [ $*MapName* ] EXPECT *Condition* ELSE *ConditionMode*;**
**TEST_ANALOG [ $*MapName* ] EXPECT *Condition* ELSE *ConditionMode*, "*ErrorMessage*";**

**Description:**

Reads the specified analog input and compares the returned value with the expected value defined in the condition part.
The value of the measured input is in mV.

**Parameters:**

| *Name* | *Value* | *Unit* | *Description* |
|---|---|---|---|
| Channel | 1...n | - | Specifies the analog input channel to be tested |
| MapName | String | - | Defined with MAP command. |
| Condition | | - | See section 5.1 on page 148 for its format. |
| ConditionMode (Optional) | IGNORE, **CONTINUE** , ABORT, ABORT_ALL | - | If the condition fails, then the condition mode indicates whether the test shall be aborted immediately or whether subsequent tests shall still be executed. By default ConditionMode = CONTINUE so if this parameter is omitted subsequent tests will be executed. |
| ErrorMessage (Optional) | String | - | When the condition fails, the specified text string is shown on the log screen |

**Example:**

```
TEST_ANALOG [ 1 ] EXPECT ( == 3.3 );
TEST_ANALOG [ 1 ] EXPECT ( == 3.3 ), "Error message 1";

TEST_ANALOG [ 307 ] EXPECT ( == 2400 ) ELSE CONTINUE;
TEST_ANALOG [ 308 ] EXPECT ( < 100 ) ELSE ABORT, "more than 100mV on
input";

TEST_ANALOG [ $Supply_12V ] EXPECT (( > 11.5 ) and ( < 12.5 ));
TEST_ANALOG [ $Supply_12V ] EXPECT (( > 11.5 ) and ( < 12.5 )), "Error
message 1";
TEST_ANALOG [ $Supply_12V ] EXPECT (( > 11.5 ) and ( < 12.5 )) ELSE
CONTINUE;
TEST_ANALOG [ $Supply_12V ] EXPECT (( > 11.5 ) and ( < 12.5 )) ELSE ABORT,
"Error message 2";
```

It is also possible to leave out the 'EXPECT' part and assign the measured value to a variable. This allows for more complex tests with an associated IF command like in the example below where two inputs are

compared.

*Note the value of 1.2. As explained on page 35, values with a decimal point indicate values in V and will be converted values in mV. Instead of 1.2, one could therefore also write 1200 in the example below.*

```
VAR #V1;
VAR #V2;

#V1 = TEST_ANALOG [ 1 ];
#V2 = TEST_ANALOG [ 2 ];

IF ( #V1 > (#V2 + 1.2) )
    FAIL "V1 exceeds V2 with more then 1.2V";
ENDIF;
```

## TEST_COUNTER*

**TEST_COUNTER [ *Channel* ];**
**TEST_COUNTER [ *Channel* ] EXPECT *Condition*;**
**TEST_COUNTER [ *Channel* ] EXPECT *Condition*, "*ErrorMessage*";**
**TEST_COUNTER [ *Channel* ] EXPECT *Condition* ELSE *ConditionMode*;**
**TEST_COUNTER [ *Channel* ] EXPECT *Condition* ELSE *ConditionMode*, "*ErrorMessage*";**

*Not available in My-TEST and eC-my-test

**Description:**

Reads the specified counter and compares the returned value with the expected value defined in the condition part.
The value that is tested depends on the configured counter mode (see CONFIG_COUNTER*). In FREQUENCY mode the value is the measured frequency in Hz while in COUNTER mode, the value is the number of detected edges (either positive or negative edges, depending on the configured EDGE).

**Parameters:**

| *Name* | *Value* | *Unit* | *Description* |
|---|---|---|---|
| Channel | 1...n | - | Specifies the counter channel to be tested. |
| Condition | | - | See section 5.1 on page 148 for its format. |
| ConditionMode<br>(Optional) | IGNORE,<br>**CONTINUE** ,<br>ABORT,<br>ABORT_ALL | - | If the condition fails, then the condition mode indicates whether the test shall be aborted immediately or whether subsequent tests shall still be executed.<br>By default ConditionMode = CONTINUE so if this parameter is omitted subsequent tests will be executed. |
| ErrorMessage<br>(Optional) | String | - | When the condition fails, the specified text string is shown on the log screen |

**Example:**

```
TEST_COUNTER [ 1 ] EXPECT ( > 0 );
TEST_COUNTER [ 1 ] EXPECT ( > 0 ), "Error message 1";
TEST_COUNTER [ 1 ] EXPECT ( > 0 ) ELSE CONTINUE;
TEST_COUNTER [ 1 ] EXPECT ( > 0 ) ELSE ABORT, "Error message 2";
```

See the CONFIG_COUNTER command for more examples.

## TEST_SERIALFIELD*

**TEST_SERIALFIELD [ *Channel* ] [ $*FieldName* ];**
**TEST_SERIALFIELD [ *Channel* ] [ $*FieldName* ] EXPECT *Condition*;**
**TEST_SERIALFIELD [ *Channel* ] [ $*FieldName* ] EXPECT *Condition*, "*ErrorMessage*";**
**TEST_SERIALFIELD [ *Channel* ] [ $*FieldName* ] EXPECT *Condition* ELSE *ConditionMode*;**
**TEST_SERIALFIELD [ *Channel* ] [ $*FieldName* ] EXPECT *Condition* ELSE *ConditionMode*, "*ErrorMessage*";**

*Not available in My-TEST and eC-my-test

**Description:**

Reads the specified packet field and compares the returned value with the expected value defined in the condition part.
Instead of simply specifying a receive packet field name, it is also possible to add a specifier in front of the field name. This specifier indicates whether the field is a field in the Rx packet or in the Tx packet. Using the 'TX.' prefix accesses the transmit packet while 'Rx.' accesses the received packet. Note that 'RX.' is optional.

**Parameters:**

| *Name* | *Value* | *Unit* | *Description* |
|---|---|---|---|
| Channel | 1...n | - | |
| FieldName | String | - | Field name as defined in Packet definitions file. |
| Condition | | - | See section 5.1 on page 148 for its format. |
| ConditionMode (Optional) | IGNORE, **CONTINUE** , ABORT, ABORT_ALL | - | If the condition fails, then the condition mode indicates whether the test shall be aborted immediately or whether subsequent tests shall still be executed. By default ConditionMode = CONTINUE so if this parameter is omitted subsequent tests will be executed. |
| ErrorMessage (Optional) | String | - | When the condition fails, the specified text string is shown on the log screen |

**Example:**

```
TEST_SERIALFIELD [ 1 ] [$RxField1] EXPECT ( == 6 );
TEST_SERIALFIELD [ 1 ] [$RxField1] EXPECT ( == 6 ), "Error message 1";
TEST_SERIALFIELD [ 1 ] [$RxField1] EXPECT ( == 6 ) ELSE CONTINUE;
TEST_SERIALFIELD [ 1 ] [$RxField1] EXPECT ( == 6 ) ELSE ABORT, "Error
message 2";
```

It possible to test on any single bit in the packet field by using the '.<n>' postfix on the field name. Testing on bit 5 of $RxField1 would be done like this:

```
TEST_SERIALFIELD [ 1 ] [$RxField1.5] EXPECT ( == 1 ), "Error message 1";
```

*Note that the lowest significant bit is bit number 0.*

The next example shows a test in which the UUT shall echo the received packet with all bytes inverted. In a FOR loop, several values are sent to the UUT and in order to check whether the received packet is correct, it is necessary to compare the received values with the values that were sent.
Using the 'TX.' prefix in front of the serial field name in the first TEST_SERIALFIELD command, the originally sent value is retrieved.

```
VAR #Temp;
VAR #Rx;

FOR #i { 0 .. 7 }
    SET_SERIALFIELD [ 1 ] $TXBYTE1 =  #i + 0x30;  // make ASCII digit
    TRANSMIT_SERIAL [ 1 ];

    //--- wait on reply from the UUT
    WAITRX [ 1 ] 500;

    #Rx = TEST_SERIALFIELD [ 1 ] [ TX.$TXBYTE1 ]; // get transmit byte

    // expected value is inverted transmit value
    #Temp = NOT #Rx;

    //--- Compare
    TEST_SERIALFIELD [ 1 ] [ $RXBYTE1 ] EXPECT  == #Temp,
         "Incorrect reply: i = #i#,  Received: #Rx# Expected: #TEMP#";
ENDFOR;
```

## TEST_TIME

**TEST_TIME;**
**TEST_TIME EXPECT *Condition*;**
**TEST_TIME EXPECT *Condition*, "*ErrorMessage*";**
**TEST_TIME EXPECT *Condition* ELSE *ConditionMode*;**
**TEST_TIME EXPECT *Condition* ELSE *ConditionMode*, "*ErrorMessage*";**

**Description:**

Compares the system time with the expected value defined in the condition part.
The system time is defined as the number of milliseconds that have elapsed since the system was started. While the absolute time value is not of much use, the number of elapsed ms between two points in the test can be: the principal usage of the command is to measure time intervals, e.g. to determine a timeout when waiting for a user action when using the SHOW_MESSAGE command or any other timeout. It is also possible to measure e.g. the interval between the activation of an output and the state of an input (see example) but accuracy will probably less then 10ms due to the Windows operating system.

The resolution of the system time depends on the system timer.

**Parameters:**

| *Name* | *Value* | *Unit* | *Description* |
|---|---|---|---|
| Condition | | - | See section 5.1 on page 148 for its format. |
| ConditionMode<br>(Optional) | IGNORE,<br>**CONTINUE** ,<br>ABORT,<br>ABORT_ALL | - | If the condition fails, then the condition mode indicates whether the test shall be aborted immediately or whether subsequent tests shall still be executed.<br>By default ConditionMode = CONTINUE so if this parameter is omitted subsequent tests will be executed. |
| ErrorMessage<br>(Optional) | String | - | When the condition fails, the specified text string is shown on the log screen |

**Example:**

Waiting for a user action. If the required result is not obtained within 10 seconds, the script shall aborted:

```
VAR #start;
VAR #responsetime = 0;
VAR #Vswitch = 0;

SHOW_MESSAGE "Activate Switch S3";
#start = TEST_TIME;

//--- Wait until the switch is toggled. The switch's state can be detected
//--- by reading analog input 5. If the voltage drops below 4V, then the
//--- switch is on.

WHILE ( #Vswitch > 4.0 AND (#responsetime - #start) < 10000)

    #Vswitch = TEST_ANALOG[5];
    #responsetime = TEST_TIME;

ENDWHILE;
```

```
HIDE_MESSAGE;

IF ( #Vswitch > 4.0 )
    FAIL "Switch was not toggled";
ENDIF;
```

## RECEIVE_SERIAL

**RECEIVE_SERIAL [ *Channel* ] "*ExpMsg*", TIMEOUT = *Timeout*;**
**RECEIVE_SERIAL [ *Channel* ] "*ExpMsg*", TIMEOUT = *Timeout ELSE ConditionMode*;**
**RECEIVE_SERIAL [ *Channel* ] "*ExpMsg*", TIMEOUT = *Timeout*, LOG = *OnOff* ;**
**RECEIVE_SERIAL [ *Channel* ] "*ExpMsg*", TIMEOUT = *Timeout ELSE ConditionMode*, LOG = *OnOff* ;**
**RECEIVE_SERIAL [ *Channel* ] "*ExpMsg*", SIZE = *Size*, TIMEOUT = *Timeout*;**
**RECEIVE_SERIAL [ *Channel* ] "*ExpMsg*", SIZE = *Size*, TIMEOUT = *Timeout ELSE ConditionMode*;**
**RECEIVE_SERIAL [ *Channel* ] "*ExpMsg*", SIZE = *Size*, TIMEOUT = *Timeout*, LOG = *OnOff* ;**
**RECEIVE_SERIAL [ *Channel* ] "*ExpMsg*", SIZE = *Size*, TIMEOUT = *Timeout ELSE ConditionMode*, LOG = *OnOff* ; = *Size*, TIMEOUT = *Timeout*, LOG = *OnOff* ;**

**Description:**

This command will process the oldest available message from receive FIFO of the specified UART channel on the Test Controller Card. The end of a message is defined by the reception of one of the configured EOL characters (See CONFIG_SERIAL) which is automatically stripped off and will not be passed to the RECEIVE_SERIAL command. All configured EOL codes will be stripped off .
RECEIVE_SERIAL is a command that is used to verify the response of the UUT to a previously sent request (with TRANSMIT_SERIAL). In the simplest case, verification may consist of checking whether the UUT responded with e.g. "OK" or "ERROR".
The received message is compared to the "ExpMsg" string that is part of the command. Besides literal text, the "ExpMsg" can also contain special pattern matching items. See a complete description in section Values in input matching strings
If the received string matches the expected string, then the command will succeed. Otherwise the command will fail and an error message is issued that either indicates a string mismatch (the received string is printed) or a timeout.
When failing, the ConditionMode determines whether the remainder of the script and test suite will be executed or not. The test will continue (but fail) if no Condition Mode is specified.

*NOTE: this command is not supported in Single Packet Mode (see Serial communications and packet definitions)*

**Parameters:**

| ***Name*** | ***Value*** | ***Unit*** | ***Description*** |
|---|---|---|---|
| Channel | 1..n | - | Specifies the channel from which data shall be received |
| ExpMsg | String | - | Sequence of characters that indicates the message that is expected to be received. Besides literal ASCII characters, it is also possible to specify values in hexadecimal format, as wildcards or with a placeholder for variables. |
| Size (Optional) | 1..n | - | Specifies the size of the expected message. Mandatory if the last CONFIG_SERIAL command specified "EOL = NONE", ignored in all other cases. |
| Timeout | 0..n | ms | Timeout period. If nothing is received withing the specified time, the command will fail.<br>Specifying 0 means the command will never timeout. |
| ConditionMode<br>(Optional) | IGNORE,<br>**CONTINUE** ,<br>ABORT,<br>ABORT_ALL | - | If the matching between received string and expected string fails, then the condition mode indicates whether the test shall be aborted immediately or whether subsequent tests shall still be executed.<br>By default ConditionMode = CONTINUE so if this parameter |

| *Name* | *Value* | *Unit* | *Description* |
|---|---|---|---|
| | | | is omitted subsequent tests will be executed. |
| OnOff (optional) | ON, **OFF** or 0...n | - | Any integer value other than 0 will be interpreted as 'ON'. Value '2' has a special meaning: all received bytes are printed in hex format, even if the byte as a printable (ASCII) value. |

**Examples:**

Give a command to the UUT to which it shall respond with "OK". The test fails if the UUT does not answer within 500 ms or answers with something else:

```
TRANSMIT_SERIAL [1] "SELFTEST";
RECEIVE_SERIAL  [1] "OK", TIMEOUT = 500;
```

Send a memory test request to the UUT, expecting back three lines. We're only interested in the last line which should be "OK" as in the example above but additionally, the remainder of the test shall be aborted:

```
TRANSMIT_SERIAL [1] "TEST EEPROM";
RECEIVE_SERIAL  [1] "*",  TIMEOUT = 1500;
RECEIVE_SERIAL  [1] "*",  TIMEOUT = 500;
RECEIVE_SERIAL  [1] "OK", TIMEOUT = 500 ELSE ABORT;
.... remainder of the test ....
```

Retrieve the value of an analog input of the UUT in order to test the reported value:

```
VAR #ADC_VAL;
TRANSMIT_SERIAL [1] "GET ADC 1";
RECEIVE_SERIAL  [1] "ADC 1 = #ADC_VAL#", TIMEOUT = 500;
IF ( #ADC_VAL < 100 OR #ADC_VAL > 110 )
    FAIL "A/D converter value is incorrect: #ADC_VAL#", ABORT;
ENDIF;
```

Retrieve a list of values and a string that contains '*' characters:

```
VAR #VAL1;
VAR #VAL2;
VAR #VAL3;
TRANSMIT_SERIAL [101] "GET VALUES";
RECEIVE_SERIAL  [101] "VALUES = #VAL1#, #VAL2#, #VAL3#", TIMEOUT = 500;
RECEIVE_SERIAL  [101] "\*\* READY \*\*", TIMEOUT = 500;
```

Also see section Terminal Mode for more advanced examples.

## RECEIVE_CAN

**RECEIVE_CAN [ *Channel* ] "*ExpMsg*", TIMEOUT = *Timeout*;**
**RECEIVE_CAN [ *Channel* ] "*ExpMsg*", TIMEOUT = *Timeout ELSE ConditionMode*;**
**RECEIVE_CAN [ *Channel* ] "*ExpMsg*", TIMEOUT = *Timeout*, LOG = *OnOff* ;**
**RECEIVE_CAN [ *Channel* ] "*ExpMsg*", TIMEOUT = *Timeout ELSE ConditionMode*, LOG = *OnOff* ;**
**RECEIVE_CAN [ *Channel* ] "*ExpMsg*", ID = *Id*, TIMEOUT = *Timeout*;**
**RECEIVE_CAN [ *Channel* ] "*ExpMsg*", ID = *Id*, TIMEOUT = *Timeout ELSE ConditionMode*;**
**RECEIVE_CAN [ *Channel* ] "*ExpMsg*", ID = *Id*, TIMEOUT = *Timeout*, LOG = *OnOff* ;**
**RECEIVE_CAN [ *Channel* ] "*ExpMsg*", ID = *Id*, TIMEOUT = *Timeout ELSE ConditionMode*, LOG = *OnOff* ;**

**Description:**

This command will process the oldest available message from the receive FIFO of the specified serial CAN channel on the Test Controller Card.
RECEIVE_CAN is a command that is used to verify the response of the UUT to a previously sent request (with TRANSMIT_CAN). In the simplest case, verification may consist of checking whether the UUT responded with e.g. "OK" or "ERROR".
The received message is compared to the "ExpMsg" string that is part of the command. Besides literal text, the "ExpMsg" can also contain special pattern matching items. See a complete description in section Values in input matching strings
If the received string matches the expected string, then the command will succeed. Otherwise the command will fail and an error message is issued that either indicates a string mismatch (the received string is printed) or a timeout.
If the ID parameter is specified then the ID of the received packet shall match this ID. If not matching, the command will fail.
When failing, the ConditionMode determines whether the remainder of the script and test suite will be executed or not. The test will continue (but fail) if no Condition Mode is specified.

If a packet is received (whether it matches the expectations or not) then information about the packet is stored in the following built-in variables:

| *Name* | *Type* | *Description* |
|---|---|---|
| #_CAN_RX_ID_ | Integer | ID of last received CAN packet |
| #_CAN_RX_DLC_ | Integer | Length of the received packet (Data Length Code field) |
| #_CAN_RX_DATA_ | Integer array | Data byte of the received packet. Array entries 0 .. #_CAN_RX_DLC_ will contain the data bytes. |

The values of these variables will remain valid until the reception of a packet with the next RECEIVE_CAN command.

Additionally, errors that occurred on the CAN bus since the last TRANSMIT_CAN command are stored in the built-in variable #_CAN_STATUS_. Every time an error occurs, the corresponding bit will be set and will not be reset until the next TRANSMIT_CAN command, even if the error condition is not present any more. The table below defines the bits in the #_CAN_STATUS_ variable:

| *Bit* | *Description* |
|---|---|
| 0 | Invalid message received |
| 1 | Undefined |
| 2 | Receiver in Error State 'Warning' |

| *Bit* | *Description* |
|---|---|
| 3 | Transmitter in Error State 'Warning' |
| 4 | Receiver in Error State 'Bus Passive' |
| 5 | Transmitter in Error State 'Bus Passive' |
| 6 | Transmitter in Error State 'Bus Off' |
| 7 | Undefined |

**Parameters:**

| *Name* | *Value* | *Unit* | *Description* |
|---|---|---|---|
| Channel | 1..n | - | Indicates which CAN channel (device) on the Test Controller Card is concerned. |
| ExpMsg | String | - | Sequence of characters that indicates the message that is expected to be received. Besides literal ASCII characters, it is also possible to specify values in hexadecimal format, as wildcards or with a placeholder for variables. |
| Id (Optional) | 1..n | - | Specifies the expected Id of the expected message. |
| Timeout | 0..n | ms | Timeout period. If nothing is received withing the specified time, the command will fail.<br>Specifying 0 means the command will never timeout. |
| ConditionMode<br>(Optional) | IGNORE, **CONTINUE** , ABORT, ABORT_ALL | - | If the matching between received string and expected string fails, then the condition mode indicates whether the test shall be aborted immediately or whether subsequent tests shall still be executed.<br>By default ConditionMode = CONTINUE so if this parameter is omitted subsequent tests will be executed. |
| OnOff<br>(optional) | ON, **OFF** or 0...n | - | Any integer value other than 0 will be interpreted as 'ON'. Value '2' has a special meaning: all received bytes are printed in hex format, even if the byte as a printable (ASCII) value. |

**Examples:**

See also RECEIVE_SERIAL and CONFIG_CAN.

The next sequence of commands will send a packet with ID = 0x100 to the UUT and expects a reply with the sameID within one second. Using the built-in variables a test is done on the length of the packet and the contents are interpreted.

```
VAR #AnswerType;
VAR #Detail;

CONFIG_CAN [ 1 ] BAUDRATE = 500000, EXTID = Off;

TRANSMIT_CAN [ 1 ] "\x00\xD5\x03\xA4", ID = 0x100, LOG = ON;

RECEIVE_CAN [ 1 ] "*", ID = 0x100, TIMEOUT = 1000, LOG = ON;
if ( #_CAN_RX_DLC_ >= 3 )
    #AnswerType = #_CAN_RX_DATA_[0];
    #Detail     = #_CAN_RX_DATA_[2];
    log "Answer type is #AnswerType#, Detail = #Detail#";
else
```

```
    FAIL "Expecting at least three bytes";
endif;
```

## RECEIVE_RS485

**RECEIVE_RS485 [ *Channel* ] "*ExpMsg*", TIMEOUT = *Timeout*;**
**RECEIVE_RS485 [ *Channel* ] "*ExpMsg*", TIMEOUT = *Timeout ELSE ConditionMode*;**
**RECEIVE_RS485 [ *Channel* ] "*ExpMsg*", TIMEOUT = *Timeout*, LOG = *OnOff* ;**
**RECEIVE_RS485 [ *Channel* ] "*ExpMsg*", TIMEOUT = *Timeout ELSE ConditionMode*, LOG = *OnOff* ;**
**RECEIVE_RS485 [ *Channel* ] "*ExpMsg*", SIZE = *Size*, TIMEOUT = *Timeout*;**
**RECEIVE_RS485 [ *Channel* ] "*ExpMsg*", SIZE = *Size*, TIMEOUT = *Timeout ELSE ConditionMode*;**
**RECEIVE_RS485 [ *Channel* ] "*ExpMsg*", SIZE = *Size*, TIMEOUT = *Timeout*, LOG = *OnOff* ;**
**RECEIVE_RS485 [ *Channel* ] "*ExpMsg*", SIZE = *Size*, TIMEOUT = *Timeout ELSE ConditionMode*, LOG = *OnOff* ;**

**Description:**

This command will process the oldest available message from the receive FIFO of the specified serial RS485 channel on the Test Controller Card. The end of a message is defined by the reception of one of the configured EOL characters (See CONFIG_RS485) which is automatically stripped off and will not be passed to the RECEIVE_RS485 command. All configured EOL codes will be stripped off .
RECEIVE_RS485 is a command that is used to verify the response of the UUT to a previously sent request (with TRANSMIT_RS485). In the simplest case, verification may consist of checking whether the UUT responded with e.g. "OK" or "ERROR".
The received message is compared to the "ExpMsg" string that is part of the command. Besides literal text, the "ExpMsg" can also contain special pattern matching items. See a complete description in section Values in input matching strings.
If the received string matches the expected string, then the command will succeed. Otherwise the command will fail and an error message is issued that either indicates a string mismatch (the received string is printed) or a timeout.
When failing, the ConditionMode determines whether the remainder of the script and test suite will be executed or not. The test will continue (but fail) if no Condition Mode is specified.

**Parameters:**

| Name | Value | Unit | Description |
|---|---|---|---|
| Channel | 1..n | - | Specifies the channel from which data shall be received |
| ExpMsg | String | - | Sequence of characters that indicates the message that is expected to be received. Besides literal ASCII characters, it is also possible to specify values in hexadecimal format, as wildcards or with a placeholder for variables. |
| Size (Optional) | 1..n | - | Specifies the size of the expected message. Mandatory if the last CONFIG_RS485 command specified "EOL = NONE", ignored in all other cases. |
| Timeout | 0..n | ms | Timeout period. If nothing is received withing the specified time, the command will fail.<br>Specifying 0 means the command will never timeout. |
| ConditionMode<br>(Optional) | IGNORE, **CONTINUE** , ABORT, ABORT_ALL | - | If the matching between received string and expected string fails, then the condition mode indicates whether the test shall be aborted immediately or whether subsequent tests shall still be executed.<br>By default ConditionMode = CONTINUE so if this parameter is omitted subsequent tests will be executed. |
| OnOff<br>(optional) | ON, **OFF** or 0...n | - | Any integer value other than 0 will be interpreted as 'ON'. Value '2' has a special meaning: all received bytes are printed in hex format, even if the byte as a printable (ASCII) |

| Name | Value | Unit | Description |
|---|---|---|---|
| | | | value. |

**Examples:**

See RECEIVE_SERIAL

## RECEIVE_COM*

**RECEIVE_ COM [ *Channel* ] "*ExpectedMessage*", SIZE = *Size*, TIMEOUT = *Timeout* LOG = *OnOff* ;**

*Not available in My-TEST and eC-my-test

**Description:**

RECEIVE_COM is a command that is used to receive and verify the response of a device to a previously sent request (with TRANSMIT_COM). The commands can be used to control a measurement device such as an oscilloscope or multimeter and receive measurements.

In the simplest case, response verification may consist of checking whether the external application responded with e.g. "OK" or "ERROR".
The received message is compared to the "ExpectedMessage" string that is part of the command. Besides literal text, the "ExpectedMessage" can also contain special pattern matching which are described in section Values in input matching strings. Also see section Terminal Mode for more advanced examples.

The end of a message is defined by the reception of a <CR> character which is automatically stripped off and will not be passed to the RECEIVE_ COM command. Other ASCII control codes (like e.g. <LF>) will also be stripped off so that the string that is passed to the command will only contain readable characters.

Before the command can be used, it is necessary to configure the channel using the CONFIG_ COM command.

**Parameters:**

| ***Name*** | ***Value*** | ***Unit*** | ***Description*** |
|---|---|---|---|
| Channel | 1...n | - | Specifies the channel from which data shall be received. See the CONFIG_TCP command. |
| ExpectedMessage | String | - | Sequence of characters that indicates the message that is expected to be received. Besides literal ASCII characters, it is also possible to specify values in hexadecimal format, as wildcards or with a placeholder for variables. |
| Size (Optional) | 1..n | - | Specifies the size of the expected message. Mandatory if the last CONFIG_COM command specified "EOL = NONE", ignored in all other cases. |
| Timeout | 0..n | ms | Timeout period. If nothing is received withing the specified time, the command will fail.<br>Specifying 0 means the command will never timeout. |
| ConditionMode<br>(Optional) | IGNORE,<br>**CONTINUE** ,<br>ABORT,<br>ABORT_ALL | - | If the matching between received string and expected string fails, then the condition mode indicates whether the test shall be aborted immediately or whether subsequent tests shall still be executed.<br>By default ConditionMode = CONTINUE so if this parameter is omitted subsequent tests will be executed. |
| OnOff<br>(optional) | ON, **OFF** or<br>0...n | - | Any integer value other than 0 will be interpreted as 'ON'. Value '2' has a special meaning: all received bytes are printed in hex format, even if the byte as a printable (ASCII) value. |

**Examples:**

tbs

## RECEIVE_TCP*

**RECEIVE_ TCP [ *Channel* ] "*ExpectedMessage*", SIZE = *Size*, TIMEOUT = *Timeout* *ELSE* *ConditionMode*, LOG = *OnOff***

*Not available in My-TEST and eC-my-test

**Description:**

RECEIVE_TCP is a command that is used to receive and verify the response of the external application to a previously sent request (with TRANSMIT_TCP). In the simplest case, verification may consist of checking whether the external application responded with e.g. "OK" or "ERROR".
The received message is compared to the "ExpectedMessage" string that is part of the command. Besides literal text, the "ExpectedMessage" can also contain special pattern matching which are described in section Values in input matching strings. Also see section Terminal Mode for more advanced examples.

The end of a message is defined by the reception of a <CR> character which is automatically stripped off and will not be passed to the RECEIVE_TCP command. Other ASCII control codes (like e.g. <LF>) will also be stripped off so that the string that is passed to the command will only contain readable characters.

Before the command can be used, it is necessary to configure the channel using the CONFIG_TCP command.

**Parameters:**

| ***Name*** | ***Value*** | ***Unit*** | ***Description*** |
|---|---|---|---|
| Channel | 1...n | - | Specifies the channel from which data shall be received. See the CONFIG_TCP command. |
| ExpectedMessage | String | - | Sequence of characters that indicates the message that is expected to be received. Besides literal ASCII characters, it is also possible to specify values in hexadecimal format, as wildcards or with a placeholder for variables. |
| Size (Optional) | 1..n | - | Specifies the size of the expected message. Mandatory if the last CONFIG_TCP command specified "EOL = NONE", ignored in all other cases. |
| Timeout | 0..n | ms | Timeout period. If nothing is received withing the specified time, the command will fail.<br>Specifying 0 means the command will never timeout. |
| ConditionMode<br>(Optional) | IGNORE,<br>**CONTINUE** ,<br>ABORT,<br>ABORT_ALL | - | If the matching between received string and expected string fails, then the condition mode indicates whether the test shall be aborted immediately or whether subsequent tests shall still be executed.<br>By default ConditionMode = CONTINUE so if this parameter is omitted subsequent tests will be executed. |
| OnOff<br>(optional) | ON, **OFF** or<br>0...n | - | Any integer value other than 0 will be interpreted as 'ON'. Value '2' has a special meaning: all received bytes are printed in hex format, even if the byte as a printable (ASCII) value. |

**Examples:**

Give a command to the external application to which it shall respond with "OK". The test fails if the application does not answer within 500 ms or answers with something else:

```
TRANSMIT_TCP [1] "SELFTEST";
RECEIVE_TCP  [1] "OK", TIMEOUT = 500;
```

Send a memory test request to an external application that is running locally. The expected answer consists of three lines. We're only interested in the last line which should be "OK" as in the example above but additionally, the remainder of the test shall be aborted:

```
CONFIG_TCP [1] PORT = 10500;
TRANSMIT_TCP [1] "TEST EEPROM";
RECEIVE_TCP  [1] "*",  TIMEOUT = 1500;
RECEIVE_TCP  [1] "*",  TIMEOUT = 500;
RECEIVE_TCP  [1] "OK", TIMEOUT = 500 ELSE ABORT;
.... remainder of the test....
```

Retrieve the value of the distortion measured by an external test device with an Ethernet connection and IP address “192.168.2.1” in order to test the reported value.
Note that the distortion is reported as a floating point value which is multiplied with 1000 before it is assigned to #DIST_VAL. This is indicated in the receive command with the ':f' after the variable name.

```
VAR #DIST_VAL;

CONFIG_TCP [1] HOST = "192.168.2.1", PORT = 10500;

TRANSMIT_TCP [1] "GET DISTORTION";
RECEIVE_TCP  [1] "DIST = #DIST_VAL:f#", TIMEOUT = 500;
IF ( #DIST_VAL > 110 )
    FAIL "Too much distortion: #DIST_VAL#", ABORT;
ENDIF;
```

Hint: debugging TCP communication can be easily done by using a freeware TCP host terminal application like 'Hercules'.
(http://www.hw-group.com/products/hercules/index_en.html)

## RECEIVE_VISA*

**RECEIVE_VISA [ *Channel* ] "*ExpectedMessage*", TIMEOUT = *Timeout* *ELSE ConditionMode*, LOG = *OnOff***

*Not available in My-TEST and eC-my-test

**Description:**

RECEIVE_VISA is a command that is used to receive and verify the response of the VISA instrument to a previously sent request (with TRANSMIT_VISA). In the simplest case, verification may consist of checking whether the VISA instrument responded with e.g. "OK" or "ERROR".
The received message is compared to the "ExpectedMessage" string that is part of the command. Besides literal text, the "ExpectedMessage" can also contain special pattern matching which are described in section Values in input matching strings.

The end of a message is defined by the reception of a <CR> and/or <LF>character which is automatically stripped off and will not be passed to the RECEIVE_VISA command.

Before the command can be used, it is necessary to configure the channel using the CONFIG_VISA command.

*Note: it is advised to keep the specified timeout as short as possible. When aborting the test while RECEIVE_VISA is waiting for a response, the command may not end before the timeout period has expired. This behavior depends on the vendor-specific implementation of the underlying VISA architecture (See also the remarks at CONFIGURE_VISA)*

**Parameters:**

| ***Name*** | ***Value*** | ***Unit*** | ***Description*** |
|---|---|---|---|
| Channel | 1...n | - | Specifies the channel from which data shall be received. See the CONFIG_VISA command. |
| ExpectedMessage | String | - | Sequence of characters that indicates the message that is expected to be received. Besides literal ASCII characters, it is also possible to specify values in hexadecimal format, as wildcards or with a placeholder for variables. |
| Timeout | 0..n | ms | Timeout period. If nothing is received withing the specified time, the command will fail.<br>Specifying 0 means the command will return immediately. |
| ConditionMode<br>(Optional) | IGNORE,<br>**CONTINUE** ,<br>ABORT,<br>ABORT_ALL | - | If the matching between received string and expected string fails, then the condition mode indicates whether the test shall be aborted immediately or whether subsequent tests shall still be executed.<br>By default ConditionMode = CONTINUE so if this parameter is omitted subsequent tests will be executed. |
| OnOff<br>(optional) | ON, **OFF** or<br>0...n | - | Any integer value other than 0 will be interpreted as 'ON'.<br>Value '2' has a special meaning: all received bytes are printed in hex format, even if the byte as a printable (ASCII) value. |

**Examples:**

The example below uses a digital voltmeter (e.g. Agilent 34401A) that is connected to COM7 to perform two voltage measurements and report the results. After some initialization commands, a trigger is given

(“TRIG:SOUR IMM” command) and the values are read back by sending a “READ?”, followed by RECEIVE_VISA in order to retrieve the two values.

```
VAR #Voltmeter = "ASRL7::INSTR";
VAR #Dvm = 1;
VAR #Val1;
VAR #Val2;

CONFIG_VISA[ #Dvm ] INSTRUMENT = #Voltmeter;
TRANSMIT_VISA[ #Dvm ] "*RST", LOG = ON; //Reset to default settings
TRANSMIT_VISA[ #Dvm ] "SYST:REM", LOG = ON; //DMM is controlled remotely
TRANSMIT_VISA[ #Dvm ] "*CLS", LOG = ON; //Clear status register
TRANSMIT_VISA[ #Dvm ] "CONF:VOLT:DC 10, 0.03", LOG = ON; //Measure DC
voltages
TRANSMIT_VISA[ #Dvm ] "TRIG:COUN 2", LOG = ON; //Dmm will accept 2 triggers
TRANSMIT_VISA[ #Dvm ] "TRIG:SOUR IMM", LOG = ON; //Trigger source is
IMMediate
WAITMS 100;           // give some time to complete the measurements
TRANSMIT_VISA[ #Dvm ] "READ?", LOG = ON; //Take readings; send to output
buffer
RECEIVE_VISA[ #Dvm ] "#Val1:f#,#Val2:f#", TIMEOUT = 500, LOG = ON;
LOG "Measured values: #Val1#mV and #Val2#mV";
```

## CALIBRATE

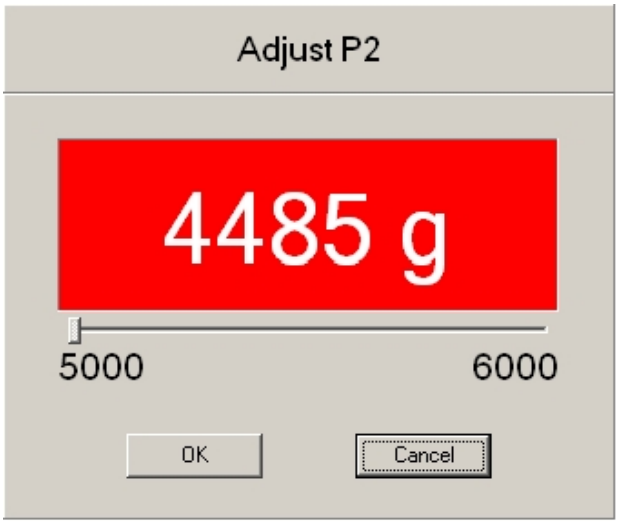
**CALIBRATE ANALOG [ *Channel* ] { *Min* .. *Max* }, Caption = "*Message*";**
**CALIBRATE ANALOG [ *Channel* ] { *Min* .. *Max* }, Caption = "*Message*" *ELSE ConditionMode*;**

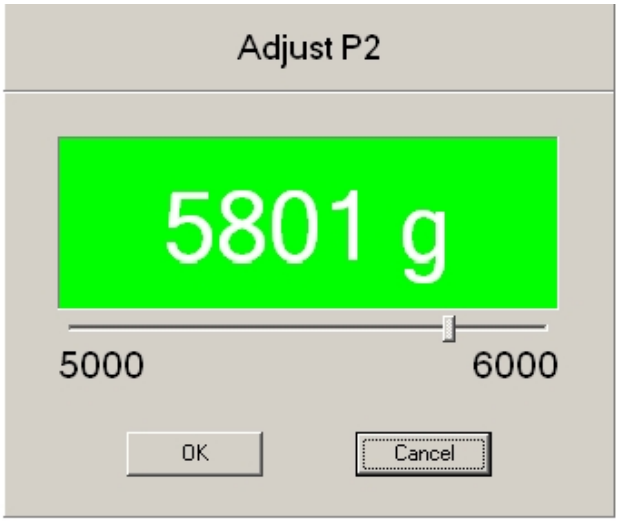
**CALIBRATE VAR [ #Variable] { *Min* .. *Max* }, Caption = "*Message*" *ELSE ConditionMode*;**
**DO**
    ***CommandList*;**
**END;**

**CALIBRATE VAR [ #Variable] { *Min* .. *Max* }, Caption = "*Message*", UNIT = "Unit" , DECIMAL = Decimal ELSE *ConditionMode*;**
**DO**
***CommandList*;**
**END;**

**Description:**

This command is used to calibrate values on the UUT. It shows a dialog box with the real-time value to be calibrated.
Min and Max specify the range in which the voltage shall lie for the calibration to be correct. As a visual help, the actual value is displayed in green when the value lies within the specified range and in red when the value lies outside the range, as shown in the two pictures below. A third element in the form of a 'LED' is shown when the value is changing outside the valid range: it is red when the value goes further away from the valid range and it is green when it approaches.




If the OK button is pressed, the actual value is checked on the valid range and determines whether the command passes or fails
The calibration value that is on the display when the user closes the dialog box is considered as an input, just like the result of any of the TEST_xxx commands. This means that the #_IN_# internal variable is set (see the section about value substitution in messages) independent of whether the user clicked 'OK' or 'Cancel'.

This first variant of this command specifies an analog input that is used as the calibration value. The second variant uses the value of the specified variable as the calibration variant. A list of commands can be specified that can do any required operation that will set the variable to a new value. The list of commands will be executed every 100ms unless the execution time of the list is more than 100ms.

This second variant is very flexible and allows to calibrate any value that requires some processing of one or more inputs.

**Parameters:**

| ***Name*** | ***Value*** | ***Unit*** | ***Description*** |
|---|---|---|---|
| Channel | 1 .. n | | Specifies the analog input channel to be tested |
| Variable | String | | Specifies the variable to be tested |
| Min | 0 .. n<br>or a variable name | - | Minimum value of the input that is acceptable. |
| Max | 0 .. n<br>or a variable name | - | Maximum value of the input that is acceptable |
| Message | String | - | The caption (title) of the dialog box |
| ConditionMode<br>(optional) | IGNORE,<br>**CONTINUE** ,<br>ABORT,<br>ABORT_ALL | - | If the condition fails, then the condition mode indicates whether the test shall be aborted immediately or whether subsequent tests shall still be executed.<br>By default ConditionMode = CONTINUE so if this parameter is omitted subsequent tests will be executed. |
| Unit (optional) | String | - | This string is displayed after the actual value and generally indicates the unit (e.g. kg or hPa) of the displayed value. |
| Decimal<br>(optional) | 0 .. n<br>or a variable name | - | Determines the position of the decimal point in the value. If, for instance, a value is measured in 0.1mV units, then a value of 1234 could be shown as '123.4 mV' by specifying Decimal = 1 and Unit = "mV" |

*Note: as usual, the Min and Max values can be specified in mV by using an integer value or in V by using a value with a decimal point.*

**Examples:**

The command is typically used to let the user adjust the value of e.g. a potentiometer in order to calibrate a value. After the user has finished, the OK button is pressed and the command determines whether the actual value is within the specified range.

The next example shows how to do a calibration and store the calibration value in the database (requires TEST-TRACK).
The test passes when the measured value lies between 200mV and 1.25V

```
VAR #Result;

#Result = CALIBRATE ANALOG [ 1 ] { 200 .. 1.25 }, CAPTION = "Potmeter 1";

Log "Calibrated on #Result:f# V, Record = ON;
```

Another interesting possibility is the use of variables for the Min and/or Max parameters. This is useful is the correct range depends on some earlier measured values. If for instance the value on a certain point must be one third of another voltage, then it is possible to do something like this (accepting a 5% deviation in the voltage):

```
MAP $Vref     ON ANALOG IN 10;
MAP $OneThird ON ANALOG IN 1

VAR #InitialVoltage;
VAR #Min;
VAR #Max;
```

```
#InitialVoltage = TEST_ANALOG [$Vref];

#Min = ((#InitialVoltage / 3) * 95)  / 100;
#Max = ((#InitialVoltage / 3) * 105) / 100;

#Result = CALIBRATE ANALOG [ $OneThird ] { #Min .. #Max }, CAPTION =
"Vref/3";

Log "Calibrated on #Result#, Record = ON;
```

This example calibrates the frequency on counter input 1. The frequency is not simply read from the counter but is divided by 10 if digital input 2 is high.
The measured frequency must be in the range 1000 .. 1200 Hz

```
VAR #Freq = 0;

Config_counter[1] Mode = FREQUENCY;

CALIBRATE VAR [ #Freq ] { 1000 .. 1200 }, CAPTION = "Frequency on counter
1", UNIT = "Hz", DECIMAL = 0
DO
    #Freq = Test_Counter [ 1 ];
    Test_Digital [2];             // implicitly sets #_IN_
    if ( #_IN_ == 1 )
          #Freq = #Freq / 10;
    endif;
END;
```

See also the example in Calibrating a frequency

## WAITMS

**WAITMS *TimeOut*;**

**Description:**

Waits for the specified number of ms.
Note that the granularity of the delayed time is 10ms. This means that the specified timeout will be rounded off to the next higher multiple of 10.
E.g. WAITMS 1234 will result in a delay of 1240 ms.

**Parameters:**

| *Name* | *Value* | *Unit* | *Description* |
|---|---|---|---|
| TimeOut | 1...n | ms | Delay period |

**Example:**

```
WAITMS 1500;
```

## WAITWHILE

**WAITWHILE ( *TimeOut* )**
**    *TestingCommand*;**
**WAITWHILE ( *TimeOut* ) Message**
**    *TestingCommand*;**
**WAITWHILE ( *TimeOut* ) ELSE *ConditionMode***
**    *TestingCommand*;**
**WAITWHILE ( *TimeOut* ) ELSE *ConditionMode*, Message**
**    *TestingCommand*;**

**Description:**

Continues executing the test command immediately following this command while this test command passes or until the specified timeout period expires.
At the end of the execution, the predefined variable #_WAITED_ is set to the time (in ms) that was spent waiting. At the same time, the variable #_IN_ reflects the last value tested by the TestingCommand.

*Note: when the test command condition fails, the loop is left but the test verdict is NOT set. As a consequence, the condition mode part of the TestingCommand (e.g. 'ELSE ABORT') is ignored. If a failure message is specified for the test command, then this message will be printed when the WAITWHILE exits with a timeout.*

The optional *Message* parameter is a message that is showed when a timeout occurs. If no message is specified, the default message indicating a timeout will be shown

*Note: the built-in variable #_ERROR_ is set to 'timeout' when a timeout occurs otherwise to 'OK'. The TestingCommand will NOT have any influence on its value.*

See #_ERROR_ : Handling errors programmatically for a detailed description of #_ERROR_.

**Parameters:**

| ***Name*** | ***Value*** | ***Unit*** | ***Description*** |
|---|---|---|---|
| TimeOut | 1…n | ms | The granularity of the time is 10 ms. This means that the specified time is e.g. 15 ms, then the actual timeout will occur after 20 ms. |
| ConditionMode<br>(Optional) | IGNORE,<br>**CONTINUE** ,<br>ABORT,<br>ABORT_ALL | - | If a timeout occurs, then the condition mode indicates whether the test shall be aborted immediately or whether subsequent tests shall still be executed.<br>By default ConditionMode = CONTINUE so if this parameter is omitted subsequent tests will be executed.<br>If mode IGNORE is specified, then the command will not cause the test to fail, even if a timeout occurs. In this case, there will also not be a message on the log screen. |
| Message | | | |
| TestingCommand | One testing command | - | The allowed commands are:<br>- All TEST_xxx commands<br>- ECEIVE_SERIAL |

**Example:**

Both examples will repeat the TEST_ANALOG command as long as the measured voltage is not 3.3V. Only when this voltage changes or after 2000 ms has passed, the loop will stop. If a timeout occurs in the first

command, then a standard message indicating 'timeout' will be printed while in the second command, the user specified message will be printed.

```
WAITWHILE ( 2000 )
    TEST_ANALOG [ 1 ] EXPECT ( != 3.3 );   // 'wait while input 1 is not
3.3V'

LOG "Input 1 reached 3.3V after #_WAITED_# ms (or #_WAITED_:f# s)";

WAITWHILE ( 2000 ) "Input 1 does not become 3.3V"
    TEST_ANALOG [ 1 ] EXPECT ( != 3.3 );

LOG "Input value of Ana 1 is #_IN_:f# V";
```

## FAIL

**FAIL "*Message*";**
**FAIL "*Message*", *ConditionMode*;**
**FAIL "*Message*", INDENT = *Value*;**
**FAIL "*Message*", *ConditionMode, INDENT* = *Value*;**

**Description:**

This command makes the test fail. The message is printed after the specified number of white spaces (zero if INDENT is omitted)..
Whether the remainder of the test script is executed or not depends on the specified condition mode. If no mode is specified, then execution will continue.

Use this command for more complex test conditions which cannot be solved by the usual TEST_xxx commands.
An example is the case where one needs to compare two measured values before the test verdict can be given. Using variables to store the different values plus an IF statement with a appropriate condition can handle such a situation. In either the IF branch or ELSE branch, the FAIL command gives the negative verdict.

See also the REPAIR command which is similar but adds a repair item to the UUT.

**Parameters:**

| *Name* | *Value* | *Unit* | *Description* |
|---|---|---|---|
| Message | String | - | This message will be displayed in the log screen as a message of type 'Failure'.<br>Just like LOG messages, variables can be used which are substituted by actual values. |
| ConditionMode<br>(Optional) | **CONTINUE**,<br>ABORT,<br>ABORT_ALL | - | If the condition fails, then the condition mode indicates whether the test shall be aborted immediately or whether subsequent tests shall still be executed.<br>By default ConditionMode = CONTINUE so if this parameter is omitted subsequent tests will be executed. |

**Example:**

**This example reads in analog input 1 with a TEST_ANALOG command that does not have an 'expect' part (it will never fail).**
**The FAIL command then uses the actually read value as part of the error message.**

```
VAR #Input;
#Input = TEST_ANALOG [1];
IF ( #Input > 3.0 )
    FAIL "The UUT does not work correctly (input = #Input# mV )";
ENDIF;
```

## FOR

**FOR** #***Variable*** **{** ***StartValue .. EndValue*** **}**
***CommandList*****;**
**ENDFOR;**

**FOR** #***Variable*** **{** ***StartValue .. EndValue*** **STEP** ***StepValue*****}**
***CommandList*****;**
**ENDFOR;**

**Description:**

Repeats the commands in the *CommandList* for each value from *StartValue* to *EndValue*. For each iteration of the loop, the value will be assigned to the variable #*Variable*.
Optionally, if 'STEP *StepValue'* is specified, then the value will be incremented by *StepValue* in each iteration, otherwise, the value is incremented by one.
Normally, the loop is repeated for all values but if the *CommandList* contains a test command which fails, then the loop is interrupted if the condition mode is 'ABORT'.

*StartValue*, *EndValue* and *StepValue* must be integer values.
=> EndValue must be greater or equal to StartValue.

**FOR** #***Variable*** **{** ***EnumeratedList*** **}**
***CommandList*****;**
**ENDFOR;**

**Description:**

This is a variation of the FOR command in which the values are not a consecutive series. Each value for which the commands in the loop shall be executed are explicitly listed. The values can be any valid expression. Note however that all values must be of the same type (e.g. all integers or all strings), otherwise an error message will occur.

**Parameters:**

| ***Name*** | ***Value*** | ***Unit*** | ***Description*** |
|---|---|---|---|
| Variable | String | - | The variable does **not** need to be declared with the usual VAR command. The validity of the variable is limited to the FOR command. This means that using the variable after the ENDFOR will result in an error. |
| Startvalue | Integer | | Initial value of *Variable*. |
| Endvalue | Integer | | Final value of *Variable* |
| Step | Integer | | Increment of *Variable* on each iteration |
| CommandList | List of commands | - | One or more commands. Any command can be used (except MAP and VAR commands) |

**Example:**

The example below will transmit three packets to the UUT with the ID field having the values 0, 1 and 2 respectively:

```
FOR #Value { 0..2 }
```

```
    SET_SERIALFIELD [ 1 ] $ID = #Value;
    TRANSMIT_SERIAL [ 1 ];
ENDFOR;
```

This example will set analog output 1 to 3.0V, 4.0V, .... 10V and then verifies whether the value read on analog input 10 is less then 400mV. Since the ABORT condition mode is specified, the loop will be interrupted (and the test as well) as soon as the measured voltage is 400mV or more. The start and end values are specified through variables. This allows for more dynamic loops.

Note that the SET_ANALOG command requires the voltage to be in mV. Not using the STEP option would result in 7000 iterations with the voltage incrementing in 1mV steps.

```
VAR #From = 3000;
VAR #To   = 10000

FOR #Value { #From .. #To STEP 1000 }
    SET_ANALOG [ 1 ] = #Value;
    TEST_ANALOG[ 10 ] EXPECT ( < 0.400 )
                      ELSE ABORT, "Current too high (#_IN_# mA)";
ENDFOR;

The next variant is applying 3.3, 5.0 and 12V to analog output 1.

FOR #Value { 3300, 5000, 12000 }
    SET_ANALOG [ 103 ] = #Value;
ENDFOR;
```

The enumerated values are also very useful for applying digital patterns to e.g. a data bus or sending multiple strings to e.g. the CAN bus. The first example below applies a pattern to an 8-bit data bus, than waits until the processor on the UUT has sent a packet back and then checks the applied pattern against the $Input field of the packet (in which the UUT has copied what it has read on the data bus):

```
MAP $Databus       ON Digital OUT Group 5, Bit 1..8;

FOR #Pattern { 0x00, 0x5A, 0x4B, 0x23, 0b10101100 }
    SET_DIGITAL [ $Databus] = #Pattern;
    WAITRX;
    TEST_SERIALFIELD[1]  [$Input] EXPECT #Pattern,
                                              “Written: #_OUT_#, Read:
#_IN_#”;
ENDFOR;
```

Also note the use of #_OUT_# and #_IN_# in the message. Whenever the test fails, a log is written in which both the last set output value (#_OUT_#, from the SET_DIGITAL command) and the last read input value (#_IN_#, from the TEST_SERIALFIELD command) are indicated.

The second example sends for messages to CAN interface 201 (the last message is an expression that appends a '?' to the values of the (string-)variable #You:

```
VAR #You = "you";
FOR #Message { "Hello", "how", "are", #You + "?" }
    TRANSMIT_CAN [201] = #Message;
ENDFOR;
```

The result will be that four packets are sent to the CAN interface: “Hello”, “how”, “are” and “You?”

## WHILE

**WHILE (** ***Condition )***
***CommandList*;**
**ENWHILE;**

**Description:**

As long as the specified *Condition* is true, the *CommandList* immediately following the condition is executed.
A *CommandList* consists of one or more commands. Any command can be used, including another WHILE command.

**Parameters:**

| *Name* | *Value* | *Unit* | *Description* |
|---|---|---|---|
| - | - | - | |

**Example:**

**This example shows how the the PWM duty cycle is incremented until a measured analog output of the UUT is 2.4V. If the output does not reach 2.4V then the test fails.**

```
VAR #InputVoltage = 0;
VAR #Duty = 10;


While ( #InputVoltage < 2.4 )
    #InputVoltage = Test_Analog[1];

    IF ( #Duty == 100 )
        FAIL "Input voltage did not reach 2.4V (actual = #InputVoltage:f#
V)";
    ENDIF;
    #Duty = #Duty + 10;
    CONFIG_PWM [1] FREQUENCY = 3000, DUTY = #Duty;
    WaitMs( 100 );   // give some time to establish the new voltage
Endwhile;
```

## IF

**IF (** ***Condition )***
***CommandList*;**
**ELIF (** ***Condition )***
***CommandList*;**
**ELSE**
***CommandList*;**
**ENDIF;**

**Description:**

The *Condition* is evaluated and if the result is true, then the *CommandList* immediately following the condition is executed.
The 'ELIF (condition) CommandList' and 'ELSE CommandList' parts are optional. ELIF can be used as many times as needed while ELSE can be used only once as the last part of the IF command.

If the optional ELSE branch is present, the commands in the *CommandList* after the ELSE keyword will be executed in case all conditions of the IF part and ELIF parts are false.
A *CommandList* consists of one or more commands. Any command can be used, including another IF command.

**Parameters:**

| ***Name*** | ***Value*** | ***Unit*** | ***Description*** |
|---|---|---|---|
| - | - | - | |

**Example:**

In its most simple form, you can simply test on one condition and act upon that condition:

```
IF ( #Voltage < 3.45 )
    SET_DIGITAL[1] = OFF;
    FAIL "Voltage too low";
ENDIF;
```

This example asks the user for a value. Depending on the value, a specific message is printed.

```
VAR #Choice;
#Choice = ASK "What is the outside temperature?", TYPE = INPUT;
IF ( # Choice < 0 )
    LOG "It is freezing";
ELIF ( # Choice < 20 )
    LOG "It is not very warm";
ELIF ( # Choice < 35 )
    LOG "Nice summer wheather";
ELSE
    LOG "it is too hot to go outside";
ENDIF;
```

## WAITTX*

**WAITTX [ *Channel* ] *Timeout* ;**
**WAITTX [ *Channel* ] *Timeout* , "*ErrorMessage*";**
**WAITTX [ *Channel* ] *Timeout* ELSE *ConditionMode*;**
**WAITTX [ *Channel* ] *Timeout* ELSE *ConditionMode*, "*ErrorMessage*";**
**WAITTX [ *Channel* ] *Timeout*, PACKETS = *Packets*;**
**WAITTX [ *Channel* ] *Timeout*, PACKETS = *Packets*, "*ErrorMessage*";**
**WAITTX [ *Channel* ] *Timeout*, PACKETS = *Packets* ELSE *ConditionMode*, "*ErrorMessage*";**

*Not available in My-TEST and eC-my-test

**Description:**

This command waits until the test engine has sent one or more packets to the UUT on the specified channel. If nothing is sent within <Timeout> milliseconds, then the command fails.

*Note that this command is only useful when the corresponding serial channel is configured with TxMode = 'Auto'.*

Typical use of this command is to assure that values just assigned to Tx Packet fields are indeed sent to the UUT before e.g. testing an output of the UUT.

When a timeout occurs, the specified message (or a standard message if none was specified) is issued indicating the failure and the built-in variable #_ERROR_ is set to 'timeout' (see #_ERROR_ : Handling errors programmatically for a detailed description of #_ERROR_)

**Parameters:**

| *Name* | *Value* | *Unit* | *Description* |
|---|---|---|---|
| Channel | 1...n | - | |
| Timeout | 1 .. n | ms | If no new packet is sent within this time, then the command will fail |
| Packets<br>(Optional) | 1 .. n | - | The number of packets that must be transmitted. Note that the timeout period restarts after each packet.<br>Default number of packets is 1. |
| ConditionMode<br>(Optional) | IGNORE,<br>**CONTINUE** ,<br>ABORT,<br>ABORT_ALL | - | On timeout, the condition mode indicates whether the test shall be aborted immediately or whether subsequent tests shall still be executed.<br>By default ConditionMode = CONTINUE so if this parameter is omitted subsequent tests will be executed. |
| ErrorMessage<br>(Optional) | String | - | On timeout, the specified text string is shown on the log screen |

**Example:**

Using the Single Packet Mode (and thus a Packet Definition file), the field $AnaOut indicates the value that the UUT shall output on its analog output. This analog output is connected to the tester's analog input 13. In order to test the UUT's output, the test shall set the $AnaOut field to the desired value and check the value on analog input 1. However, since the serial interface is configured with TxMode = AUTO, it is necessary wait with the check until the UUT did actually receive a packet. This can be done with a WAITMS command but the disadvantage is that it may either wait too shortly or that it may wait unnecessarily long. A better solution is to synchronize on the actual transmission with the WAITTX command:

```
MAP $ANA_READBACK ON ANALOG IN 13;

SET_SERIALFIELD [1] $AnaOut = 2345;
WAITTX [1] 500;
TEST_ANALOG [$ANA_READBACK] EXPECT (> 2340 AND < 2350),
                                        "Output not set correctly (#_IN_#
mV)";
```

## WAITRX*

**WAITRX [ *Channel* ] *Timeout* ;**
**WAITRX [ *Channel* ] *Timeout* , "*ErrorMessage*";**
**WAITRX [ *Channel* ] *Timeout* ELSE *ConditionMode*;**
**WAITRX [ *Channel* ] *Timeout* ELSE *ConditionMode*, "*ErrorMessage*";**

*Not available in My-TEST and eC-my-test

**Description:**

This command is only useful in Single Packet Mode in which the actual handling of the received packet is done in the background, not requiring any intervention from the script.

The command waits until a new packet (or several, if the PACKETS parameter is specified) is received on the specified channel. Packets with a different length than the length specified in the Packet Definition File (see Serial communications and packet definitions) will be ignored. This also applies to packets whose checksum is not correct (if the checksum function is used in the Packet Definition).
The command allows synchronization of commands that test received packet contents with the reception of a packet, assuring e.g. that the packet was sent after the execution of earlier commands. See the examples.
When a timeout occurs, the specified message (or a standard message if none was specified) is issued indicating the failure and the built-in variable #_ERROR_ is set to 'timeout' (see #_ERROR_ : Handling errors programmatically for a detailed description of #_ERROR_)

**Parameters:**

| *Name* | *Value* | *Unit* | *Description* |
|---|---|---|---|
| Channel | 1 .. n | - | |
| Timeout | 1 .. n | ms | When no new packet is received within this time, then the command will fail. |
| Packets<br>(Optional) | 1 .. n | - | The number of packets that must be received. Note that the timeout period restarts after each packet.<br>Default number of packets is 1. |
| ConditionMode<br>(Optional) | IGNORE,<br>**CONTINUE** ,<br>ABORT,<br>ABORT_ALL | - | On timeout, the condition mode indicates whether the test shall be aborted immediately or whether subsequent tests shall still be executed.<br>By default ConditionMode = CONTINUE so if this parameter is omitted subsequent tests will be executed. |
| ErrorMessage<br>(Optional) | String | - | On timeout, the specified text string is shown on the log screen |

**Example:**

The WaitRx commands is useful when the UUT reports back some status that depend on either inputs of the UUT or data in a serial packet sent to the UUT.
Supposing a UUT that has an analog input that must be tested, the commands to be used will first set a value on the analog input (which is connected to analog output 6 of the tester) and shall then verify the contents of the $AnaIn field of the packet that the UUT sends to the tester.
The problem is that since packets are sent/received in the background it is hard to predict when the receive packet field represents data that is received AFTER the analog input was stimulated. Using WAITMS is a solution but it will either wait not long enough or it may wait unnecessarily long.
A better solution is to synchronize on the actual reception with the WAITRX command:

```
MAP $ANA_In ON ANALOG OUT 6;
```

```
SET_ANALOG [$ ANA_In] = 3.7;
WAITRX [1] 500;
TEST_SERIALFIELD [1] [$AnaIn] EXPECT EXPECT (>3.6 AND < 3.8);
```

## Configuration

### MAP

**MAP $*MapName* ON DIGITAL IN BIT *BitRange*;**
**MAP $*MapName* ON DIGITAL OUT GROUP *GroupNumber*, BIT *BitRange*;**
**MAP $*MapName* ON ANALOG IN *Channel*;**
**MAP $*MapName* ON ANALOG OUT *Channel*;**

**Description:**

Groups one or more digital/analog inputs or outputs into a new name.
The inputs and outputs provided by the TEST-OK test bench are numbered (e.g. analog output 1 ..16, or digital input 1 ..24). When writing a test, the test author will rather think of the UUT's functions and signal names instead of the numbered I/O. The MAP command is provided to assign a name to an input or output or, in case of digital input or outputs, to a group of inputs or outputs.

*It is good practice to include all MAP commands in the preamble script that is executed before every test in TEST-TRACK, My-TEST and eC-my-test.*

It is not allowed to redefine an existing MapName. An error is generated and the script is aborted if an attempt to redefine is made.

**Parameters:**

| ***Name*** | ***Value*** | ***Unit*** | ***Description*** |
|---|---|---|---|
| $MapName | String starting with $ | - | Characters allowed in the string are upper and lower case letters, numbers and '_'. |
| BitRange | Range of integer: first..last | - | Describes the *first* and *last* bit that will be part of the mapping. Note that the lowest bit number is 1.<br>Note that *last* must be greater than *first* |
| GroupNumber | Integer | - | |
| Channel | 1...n | - | |

**Example:**

Four digital outputs that are connected to a data bus on the UUT can be assigned a name like 'DataIn':

```
MAP $DataIn       ON  DIGITAL IN     BIT 1..4;
```

Setting this data bus to the value '0101' would then be very simple and comprehensive:

```
SET_DIGITAL [ $DataIn ] = 0b0101;
```

Other examples, setting a set of input/output bits or an analog input or output:

```
MAP $5V_CentralBus          ON Analog IN 1;
MAP $Vcc                    ON ANALOG IN 101;
MAP $Potmeter1              ON ANALOG IN 102;
MAP $Potmeter2              ON ANALOG IN 103;
MAP $Ntc                    ON ANALOG IN 104;
MAP $FuehlerAufBauPos       ON Analog OUT 7;
```

```
MAP $Vout_24V              ON ANALOG OUT 101;
MAP $WaterLevel             ON Digital OUT Group 2, Bit 5..8;

MAP $PWM_1                  ON DIGITAL IN BIT 101;
MAP $BUTTON                ON DIGITAL IN BIT 103;
```

## CONFIG_SERIAL

```
CONFIG_SERIAL [ Channel ]
        BAUDRATE        = BaudRate,
        PARITY          = Parity,
        TXMODE          = TxMode,
        RXMODE          = RxMode,
        RXTHRESHOLD  = RxThreshold,
        RXTIMEOUT       = RxTimeout,
        STXMODE       = StxMode,
        EOL             = EOL;
```

**Description:**

Configures the specified serial (UART) channel on the Test Controller Card.
Depending on the mode that is chosen for serial communication (see chapter Serial communications and packet definitions), some of the parameters must be set to specific values or do not have any effect.
Therefore, there are two parameter description tables, one for each mode.

*Note that Single Packet Mode is activated by specifying a Packet Definition file. (Not supported in My-TEST and eC-my-test).*

**Parameters:**

Valid for all modes:

| *Name* | *Value* | *Unit* | *Description* |
|---|---|---|---|
| Channel | 1...n | - | Indicates which channel on the Test Controller Card is concerned. |
| BaudRate | 4800 .. 38400 | Baud | This is the baud rate that will be used on the channel. Although any value in the specified range can be given , there is a chance that the actual baud rate will slightly differ from the specified one if 'non-standard' values are used |
| Parity | EVEN, ODD, NONE | - | Enables/disables a 9-th parity bit. |
| RxThreshold | 1..n | Bytes | This value determines the size of the received packet. If The specified number of bytes is received from the UUT, it is considered as a complete packet and will be handled by the test engine.<br>If StxMode is on (see below), then this value is ignored. |
| RxTimeout | 1..n | Ms | Another way for the test engine to determine the end of a received packet. If no data is received during the specified interval, then the reception of a complete packet is assumed and will be handled by the test engine.<br>If StxMode is on (see below), then this value is ignored. |

Single Packet Mode (Not supported in My-TEST and eC-my-test):

| *Name* | *Value* | *Unit* | *Description* |
|---|---|---|---|
| TxMode | AUTO, | - | If AUTO is specified, then the test engine will continuously send about |

| *Name* | *Value* | *Unit* | *Description* |
|---|---|---|---|
| | MANUAL | | 5 packets per second to the UUT. In MANUAL mode, packets are only sent when explicitly requested by a TRANSMIT_SERIAL command |
| RxMode | AUTO | - | Only AUTO is supported. Received packets continuously update the fields that can be accessed with the TEST_SERIAL command. |
| StxMode | ON, OFF | - | If ON, then the test engine will interpret special STX, ETX and DLE bytes in order to determine the start and end of a packet. See the explanation of the mechanism in chapter 6 |
| Eol | NONE, CR, LF, CRLF | - | Ignored (this parameter is optional) |

Terminal Mode:

| *Name* | *Value* | *Unit* | *Description* |
|---|---|---|---|
| TxMode | MANUAL | - | Only MANUAL is supported. Packets are only sent when executing a TRANSMIT_SERIAL command |
| RxMode | MANUAL | - | Only MANUAL is supported. Received packets are handles by a RECEIVE_SERIAL command |
| StxMode | OFF | - | In Terminal mode, all bytes are ASCII and no special characters can be used. |
| Eol | NONE, CR, CR_RX_ANY, LF, LF_RX_ANY, **CRLF** | - | Determines whether strings sent in terminal mode are terminated by a single CR, a single LF, both CR + LF characters or nothing at all.<br>The parameter is optional. If not specified, CRLF is used.<br>In case CRLF is specified, RECEIVE_SERIAL also accepts packets that have a single CR or LF character.<br>If NONE is specified, then the RECEIVE_SERIAL command must specify the size of the expected packet.<br>Also see Table 1: EOL parameter values below |

| *EOL parameter* | *Transmitted packets* | *Received packets* |
|---|---|---|
| CRLF | CR + LF | CR, LF or both |
| CR | CR | CR |
| CR_RX_ANY | CR | CR, LF or both |
| LF | LF | LF |
| LF_RX_ANY | LF | CR, LF or both |
| NONE | - | Size parameter determines packet length |

Table 1: EOL parameter values

**Example:**

Using automatic ('background') transmission and reception in Single Packet Mode, a script to set a field in the transmitted packet and to test on a value in an incoming packet may look like this:

```
CONFIG_SERIAL [1] BaudRate = 19200, Parity = NONE, TxMode = AUTO, RxMode =
```

```
AUTO, RxThreshold = 32, RxTimeout = 10, StxMode = OFF;

SET_SERIAL [1] $LEDS_OUT = 0b10010000;
WAITRX [1] 500;
TEST_SERIALFIELD [1] [$ANALOG_IN] EXPECT > 100;
```

*Note the WAITRX command that assures that at least one new packet is received from the UUT after the SET_SERIAL was executed.*

When using the Terminal mode, the script may look like this:

```
VAR #ANA_IN;

CONFIG_SERIAL [101] BaudRate = 19200, Parity = NONE, TxMode = MANUAL,
RxMode = MANUAL, RxThreshold = 32, RxTimeout = 10, StxMode = OFF, Eol =
CRLF;

TRANSMIT_SERIAL [101] "SET LEDS: 0x90";
RECEIVE_SERIAL [101] "ANALOG = #ANA_IN#", TIMEOUT = 100;
IF ( #ANA_IN <= 100 )
    FAIL "Analog value incorrect (#ANA_IN#)";
ENDIF;
```

## CONFIG_CAN

**CONFIG_CAN [ Channel ] BAUDRATE = *BaudRate*, EXTID = *OnOff*;**

**Description:**

Configures the specified CAN channel on the Test Controller Card. Supported baud rates depend on the Test Controller that is used. Refer to the appropriate reference manual for more information.
If the EXTID parameter is 'ON' (or any non-zero value), then all transmitted packets will use extended (29-bit) ID's. Otherwise, standard (11-bit) ID's.

**Parameters:**

| *Name* | *Value* | *Unit* | *Description* |
|---|---|---|---|
| Channel | 1...n | - | Indicates which CAN channel (device) on the Test Controller Card is concerned. |
| BaudRate | integer | Bit/s | This is the baud rate that will be used on the channel. Supported baud rates are specified in the Hardware Reference Manual of the Test Controller Cards that is used. |
| ExtId | ON, **OFF** or 0...n | - | Determines whether transmitted packets use the standard or extended ID.<br>Any integer value other than 0 will be interpreted as 'ON'. |

**Example:**

The next commands will configure CAN interface 1 at 250 kbps and will use extended IDs for transmission. A 4-byte packet with ID = 0x56 containing 0x00 0xD5 0x03 and 0xA4 will be send to the UUT. We then expect an answer with ID = 0x55 (ignoring the packet contents due to the '*" wildcard) followed by a packet containing the 7 characters “TEST OK”.
The LOG command prints out the ID of the last received packet using the built-in variable #_CAN_RX_ID_.

```
CONFIG_CAN [ 1 ] BAUDRATE = 250000, EXTID = ON;
TRANSMIT_CAN [ 1 ] "\x00\xD5\x03\xA4", ID = 0x56, LOG = ON;
RECEIVE_CAN [ 1 ] "*", ID = 0x55, TIMEOUT = 10000, LOG = ON;
RECEIVE_CAN [ 1 ] "TEST OK", TIMEOUT = 10000, LOG = ON;
LOG "Received ID = #_CAN_RX_ID_:x#";
```

## CONFIG_RS485

```
CONFIG_RS485 [ Channel ]
        BAUDRATE        = BaudRate,
        PARITY          = Parity,
        RXTHRESHOLD  = RxThreshold,
        RXTIMEOUT       = RxTimeout,
        EOL             = EOL;
```

**Description:**

Configures the specified RS485 channel on the Test Controller Card.

**Parameters:**

| *Name* | *Value* | *Unit* | *Description* |
|---|---|---|---|
| Channel | 1...n | - | Indicates which channel on the Test Controller Card is concerned. |
| BaudRate | 4800 .. 38400 | Baud | This is the baud rate that will be used on the channel. Although any value in the specified range can be given , there is a chance that the actual baud rate will slightly differ from the specified one if 'non-standard' values are used |
| Parity | EVEN, ODD, NONE | - | Enables/disables a 9-th parity bit. |
| RxThreshold | 1..n | Bytes | This value determines the size of the received packet. If The specified number of bytes is received from the UUT, it is considered as a complete packet and will be handled by the test engine.<br>If StxMode is on (see below), then this value is ignored. |
| RxTimeout | 1..n | Ms | Another way for the test engine to determine the end of a received packet. If no data is received during the specified interval, then the reception of a complete packet is assumed and will be handled by the test engine.<br>If StxMode is on (see below), then this value is ignored. |
| Eol | NONE, CR, CR_RX_ANY, LF, LF_RX_ANY, **CRLF** | - | Determines whether strings sent in terminal mode are terminated by a single CR, a single LF, both CR + LF characters or nothing at all.<br>The parameter is optional. If not specified, CRLF is used.<br>In case CRLF is specified, RECEIVE_RS485 also accepts packets that have a single CR or LF character.<br>If NONE is specified, then the RECEIVE_RS485 command must specify the size of the expected packet.<br>Also see Table 2: EOL parameter values below |

| *EOL parameter* | *Transmitted packets* | *Received packets* |
|---|---|---|
| CRLF | CR + LF | CR, LF or both |
| CR | CR | CR |
| CR_RX_ANY | CR | CR, LF or both |
| LF | LF | LF |

| *EOL parameter* | *Transmitted packets* | *Received packets* |
|---|---|---|
| LF_RX_ANY | LF | CR, LF or both |
| NONE | - | Size parameter determines packet length |

Table 2: EOL parameter values

**Example:**

## CONFIG_DIGITAL_GROUP

**CONFIG_DIGITAL_GROUP [ GroupNumber ] = Value;**

**Description:**

Sets the '1' level output voltage for the specified group (note that specifying the voltage for open collector/ open drain outputs will have no effect).

**Parameters:**

| ***Name*** | ***Value*** | ***Unit*** | ***Description*** |
|---|---|---|---|
| GroupNumber | Integer | - | Selects the digital output group to be configured |
| Value | Integer or Float | mV or V | The value expresses a voltage. A constant value may either contain a decimal point or not. If there is a decimal point, then the value is in volts. Otherwise the value is in mV. When using a variable, then the value is always in mV. |

**Example:**

**The next command will set group 3 for a 3.3V digital interface.**

```
CONFIG_DIGITAL_GROUP [ 3 ] = 3.300;
```

## CONFIG_PWM*

**CONFIG_PWM [ *Channel* ] FREQUENCY = *Frequency*, DUTY = *DutyCycle*;**

*Not available in My-TEST and eC-my-test

**Description:**

Configures the specified PWM channel.
Check the reference manual of your Test Controller Card to find out how the output voltage during the 'active' phase of the PWM wave form is set.

**Parameters:**

| ***Name*** | ***Value*** | ***Unit*** | ***Description*** |
|---|---|---|---|
| Channel | 1...n | - | Indicates the specific PWM channel that must be configured |
| Frequency | integer | Hz | The frequency of the PWM signal. The actual frequency range depends on the connected Test Controller Card. |
| DutyCycle | 0..100 | % | The proportion of time during which the PWM output is high |

**Example:**

```
CONFIG_PWM [1] FREQUENCY = 3000, DUTY = 41;
```

## CONFIG_COUNTER*

**CONFIG_COUNTER [ *Channel* ] MODE = FREQUENCY;**
**CONFIG_COUNTER [ *Channel* ] MODE = COUNT, EDGE = *Edge* ;**
**CONFIG_COUNTER [ *Channel* ] MODE = PULSEWIDTH, EDGE = *Edge* ;**

*Not available in My-TEST and eC-my-test

**Description:**

Sets the counter to Count or Frequency mode.
In Count mode, the number of positive or negative edges is counted. On configuration, the counter is reset. Whether positive or negative edges are counted depends on the Edge parameter.
In PulseWidth mode, the pulse width of positive or negative pulses is measured.
In Frequency mode, the frequency of the signal on the counter input is measured. In this mode, the Edge parameter is ignored and need not to be specified.

*Note: Depending on the Test Controller Card that is used, there may be some restrictions on each of the modes and especially if more than one channel is configured during the same test. Refer to the Hardware Reference Manual of used Test Controller Card that is controlled.*

**Parameters:**

| ***Name*** | ***Value*** | ***Unit*** | ***Description*** |
|---|---|---|---|
| Channel | 1...n | - | Selects the counter to be configured |
| Mode | FREQUENCY, COUNTER, PULSEWIDTH | - | Specifies the mode of the counter:<br>· FREQUENCY: the frequency of the signal on the counter input is measured in Hz<br>· COUNTER: the counter counts the number of positive or negative edges (see Edge parameter) on the counter input.<br>· PULSEWIDTH: measures the width of positive / negative pulses on the counter channel in µs. |
| Edge | POS, NEG | | Only valid in COUNTER mode, this parameter indicates whether positive edges or negative edges shall be counted. |

**Example:**

Verify whether the frequency on counter channel is between 100 and 120 Hz:

```
CONFIG_COUNTER [1] MODE = FREQUENCY;
TEST_COUNTER [1] EXPECT (>= 100 AND <= 120), "Incorrect frequency of #_IN_#
Hz";
```

Count the number of negative edges on counter channel 2 as long as analog input 1 is below 1.25V:

```
VAR #Pulses = 0;

CONFIG_COUNTER [2] MODE = COUNTER, EDGE = NEG;
//--- note that this is not necessary just after configuration
RESET_COUNTER [2];

WAITWHILE (10000)
    TEST_ANALOG [1] EXPECT < 1.25;
```

```
#Pulses = TEST_COUNTER [2];
```

Calibrate the duty cycle on counter input 2:

```
VAR #Pos = 0;
VAR #Neg = 0;
VAR #Duty = 0;

CALIBRATE VAR[ #Duty ] { 10 .. 20 }, CAPTION = "Duty Cycle on counter 2" DO

    CONFIG_COUNTER [ 2 ] MODE = PULSEWIDTH, EDGE = POS;
    WAITMS 100;
    #Pos = TEST_COUNTER [ 2 ];
    CONFIG_COUNTER [ 2 ] MODE = PULSEWIDTH, EDGE = NEG;
    WAITMS 100;
    #Neg = TEST_COUNTER [ 2 ];
    #Duty =  (100 * #Pos) / ( #Pos + #Neg );

END;
```

The TCC1800 cannot measure pulses shorter than 10 µs: it misses the first terminating edge and reports the time until the second terminating edge.
Suppose the positive pulse is too short. The TCC1800 misses the first falling edge and the reported time is:

$$T_{report} = T_{period} + T_{pos}$$

Since $T_{period} = T_{neg} + T_{pos}$, we see that:

$T_{report} = T_{neg} + T_{pos} + T_{pos}$

Thus, $T_{pos} = (T_{report} - T_{neg}) / 2$

If the frequency of the signal is approximately known then it is possible to derive the correct pulse width from the measured results (without using the frequency!) by simply comparing $T_{neg} + T_{pos}$ with the expected $T_{period}$.

The script below does exactly that, knowing that the frequency is about 12.2 kHz (82 µs period):

```
Var #Tpos;
Var #Tneg;

Config_Counter[2] Mode = Pulsewidth, Edge = Pos;
Waitms 200;          // give some time to do the measurement
#Tpos = Test_Counter[2];

Config_Counter[2] Mode = Pulsewidth, Edge = Neg;
Waitms 200;
#Tneg = Test_Counter[2];

if ( #Tneg + #Tpos > 90 )   // more than 10% longer than the actual
frequency => one of the pulses is too small to be measured
    if ( #Tpos > #Tneg )
       #Tpos = (#Tpos - #Tneg) / 2;
    else
       if ( #Tneg > #Tpos )
          #Tneg = ( #Tneg - #Tpos) / 2;
       endif;
    endif;
endif;
```

```
#Frequency = 1000000 / (#Tpos + #Tneg);

Log "F = #Frequency#, #Tpos# us high, #Tneg# us low";
```

## CONFIG_SUPPLY

**CONFIG_SUPPLY [ *Channel* ] VOLTAGE = *Voltage*, CURRENTLIMIT =*Current*;**

**Description:**

Sets the voltage and current limit of the specified supply. If the actual current exceeds the specified current limit, then the supply will switch off and report an overload error which will abort the execution of the test suite. Before the power supply can be switched on again, it shall first be reset using the SET_SUPPLY[n] = OFF command.

*By specifying 0 for the current limit, the limiter function is disabled and the current will be limited to the maximum value the associated power supply can deliver.*

**Parameters:**

| ***Name*** | ***Value*** | ***Unit*** | ***Description*** |
|---|---|---|---|
| Channel | 1...n | - | Selects the power supply to be configured |
| Voltage | Float | V | Sets the output voltage of the power supply |
| Current | Float | A | Sets the current limit of the power supply. When the actual current exceeds this value, then the power supply will automatically shut down (see the specific details of each power supply in the TEST-OK test bench' manual |

**Example:**

```
VAR #MaxCurrent = 0.200;

CONFIG_SUPPLY [ 1 ] VOLTAGE =  5.000, CURRENTLIMIT = #MaxCurrent;
```

## CONFIG_PROGRAMMER*

**CONFIG_PROGRAMMER [ *Channel* ] "*DeviceName*";**
**CONFIG_PROGRAMMER [ *Channel* ] "*DeviceName*", MODE = *Mode*;**
**CONFIG_PROGRAMMER [ *Channel* ] "*DeviceName*", SUPPLY = *SupplyChannel*;**
**CONFIG_PROGRAMMER [ *Channel* ] "*DeviceName*", MODE = *Mode*, SUPPLY = *SupplyChannel*;**
**CONFIG_PROGRAMMER [ *Channel* ] "*DeviceName*", SPEED = *Speed*;**
**CONFIG_PROGRAMMER [ *Channel* ] "*DeviceName*", MODE = *Mode, SPEED* = *Speed*;**
**CONFIG_PROGRAMMER [ *Channel* ] "*DeviceName*", SUPPLY = *SupplyChannel, SPEED* = *Speed*;**
**CONFIG_PROGRAMMER [ *Channel* ] "*DeviceName*", MODE = *Mode*, SUPPLY = *SupplyChannel, SPEED* = *Speed*;**

**CONFIG_PROGRAMMER [ *0* ] "*DeviceName*", SERIAL = *SerialChannel*,**
**RESET = *$ResetPin* , BOOT = *$BootPin* ,**
**XTAL = *Frequency;***

*Not available in My-TEST and eC-my-test

**Description:**

Configures the specified programmer for a specific device name.

The first four variants concern the physical programmers on the TCC board while the third variant concerns programming via a serial interface of the TCC board. This will only work for devices that have a built-in boot loader.

Note that this command only provides information to the Test Engine. Nothing is done towards the device to be programmed. See the PROGRAM command for the actual programming action.

**Physical programmers**
Due to the flexible architecture of the programmer hardware many microcontrollers can be programmed using this interface.
Some devices (like the PIC12F629) require that the Vdd of the chip must be switched off at certain stages of programming, this mode is call 'Vpp before Vdd' since Vpp must be applied first. For such devices, it is necessary to specify *Mode* = *VPP*_FIRST with the additional <SupplyChannel> parameter indicating the channel of the Power Supply that supplies the device (this may be indirect via e.g. a voltage regulator on the UUT).
The Test Engine will give an error message when no supply channel is specified for a device that requires one or when the device does not support the specified mode.
It is also possible to let the programmer generate slower clock pulses in cases where additional electronics on the programming pins cause bad shaped pulses (e.g. due to a FET gate or series resistors). In the case, the SPEED can be set to 'SLOW'.

**Bootloader programmers**
Devices like NXP's LPC2xxx series of microcontrollers have a built-in boot loader that allows to load an application program via a serial interface. To enter boot mode, it is generally necessary to activate a dedicated 'boot mode' pin while applying a reset to the device. Therefore, the command requires the specification to which output pins the Reset and Boot Mode pins are connected. Generally, this should be open collector pins but any output pin can be specified.
Both outputs are specified as a MAP name. See the MAP command on how to define a MAP name for a digital output.
For some devices, it is also necessary to specify the X-Tal frequency on which the UUT Is running.
It is necessary to use the CONFIG_SERIAL command to setup the serial channel to the correct baud rate before the programming is started. Besides the baud rate, the other parameters shall be configured as follows:

| *Name* | *Value* |
|---|---|
| Parity | None |
| TxMode | Manual |
| RxMode | Manual |
| RxThreshold | 100 |
| RxTimeout | 2 |
| StxMode | Off |

CONFIG_SERIAL settings for boot mode programming.

**Supported devices**

Please contact a TEST-OK sales representative for more information about the ever growing list of supported devices.

**Parameters:**

| *Name* | *Value* | *Unit* | *Description* |
|---|---|---|---|
| Channel | 0...n | - | Selects the programmer to be configured. The number of available programmers may vary between TEST-OK test benches.<br>NOTE: Channel 0 is the 'boot loader' channel. Other channels are physical programmers on the TCC board. |
| DeviceName | String | - | Microchip PIC device name. |
| Mode (optional) | **VPP_FIRST**, VDD_FIRST | - | Defines the mode that is used to enter programming mode of a PIC device. If VDD_FIRST is specified, then it is necessary to specify a *SupplyChannel* as wel |
| Speed (optional) | **NORMAL**, SLOW | - | If set to SLOW, the clock pulses generated by the programmer will be longer. This option is useful if the programming pins of the processor are shared with other electronics that cause the signal edges to be deformed. |
| SupplyChannel | 1..n | - | Supply that is used to provide power to the device to be programmed. Mandatory when *Mode* is VDD_FIRST. |
| SerialChannel | 1..n | | Serial channel of the TCC board to which the UUT is connected |
| ResetPin | String | | Name of a MAP variable specifying the digital output to which the RESET pin of the processor is connected |
| BootPin | String | | Name of a MAP variable specifying the digital output to which the 'Boot mode' pin of the processor is connected |
| Frequency | 1..n | kHz | X-Tal frequency of the device to be programmed |

**Examples:**

Programming a PIC18F4620:

```
CONFIG_PROGRAMMER [1] "PIC18F4620";
PROGRAM [1] = "MyHexFile.hex";
```

Programming a PIC12F1822 via the Vdd before Vpp program entry mode and which is powered by supply 2:

```
CONFIG_PROGRAMMER [1] "PIC12F1822", MODE = VDD_FIRST, SUPPLY = 2;
PROGRAM [1] = "MyHexFile.hex";
```

Programming a PIC16F876 with 4K7 serial resistors in the PGD and PGC lines:

```
CONFIG_PROGRAMMER [1] "PIC16F876", SPEED = SLOW;
PROGRAM [1] = "MyHexFile.hex";
```

Programming an LPC2148 on 38400 baud:

```
CONFIG_SERIAL [1] BaudRate = 38400,
                     Parity = NONE,
                  TxMode = MANUAL,
                  RxMode = MANUAL,
                  RxThreshold = 100,
                  RxTimeout = 5,
                  StxMode = OFF;

CONFIG_PROGRAMMER [ 0 ] "LPC2148", SERIAL = 1,
                                   RESET = 1, BOOT = 2,
                                   XTAL = 12000;
PROGRAM [0] = "MyHexFile.hex";
```

## CONFIG_EXTIO*

**CONFIG_EXTIO IN = *InputBytes*, OUT = *OutputBytes*;**

*Not available in My-TEST and eC-my-test

**Description:**

Defines how much bytes can be read from the extended I/O interface on the Test Controller Card (TCC) and how many bytes can be written to it.
In order to extend the number of available inputs and outputs of the TCC, an extension interface has been provided to/from which a number of bytes can be read.
Using the CONFIG_EXTIO command lets the system know how much data shall be written and read.

The OUT bytes will be available as new digital output groups. If the TCC board that is used has n digital output groups, then the specification of two OUT bytes will result in group n+1 and group n+2 (each having 8 bits).
The IN bytes will be available as new digital inputs. If the TCC board that is used has m digital inputs, then the specification of three IN bytes will result in digital inputs m+1 .. m+24.

**Parameters:**

| *Name* | *Value* | *Unit* | *Description* |
|---|---|---|---|
| InputBytes | 0..n | - | Number of extra input bytes available |
| OutputBytes | 0..n | - | Number of extra output bytes available |

**Example:**

On the TEST-OK module that is connected to a TCC1800 board, which has 6 digital output groups and 24 digital inputs, two output shift registers and one input shift registers are placed.
The following code will (endlessly!) drive one of two LEDs that are connected to the last two output bits of the second output shift register. Depending on the value on the second input of the first input shift register the first or second led will be driven:

```
CONFIG_EXTIO IN = 1, OUT = 2;
MAP $Input      ON DIGITAL IN  BIT 26;          // first extended input
starts at 25
MAP $Led1       ON DIGITAL OUT GROUP 8, BIT 7;
MAP $Led2       ON DIGITAL OUT GROUP 8, BIT 8;

VAR #Input;

WHILE ( 1 == 1 )
  #Input = TEST_DIGITAL[$Input];
  SET_DIGITAL [$Led1] = !#Input;
  SET_DIGITAL [$Led2] =  #Input;
ENDWHILE;
```

## CONFIG_COM*

| |
|---|
| **CONFIG_COM [ *Channel* ] BAUDRATE = *BaudRate*, EOL = *Eol;*** |
| **CONFIG_COM [ *Channel* ] BAUDRATE = *BaudRate*, DATABITS = *DataBits*, PARITY = *Parity*, STOPBITS = *StopBits*, EOL = *Eol;*** |
| **CONFIG_COM [ *Channel* ] CLOSE;** |

*Not available in My-TEST and eC-my-test

**Description:**

The first two versions open and configure a COM port for communication. After having done this, it is possible to communicate with the application via dedicated transmit and receive commands (see TRANSMIT_COM and RECEIVE_COM).
The last version closes a previously opened port. Note that closing of the port is automatically done by the TestEngine at the end of the test so the CLOSE version is only needed when the port must be closed during the execution of the test.

*Note: it is possible to use the command several times in the same script. Each time a configuration is specified (the first two command versions) the COM port will be closed and reopened. Besides for changing the communication parameters, this feature is also useful when communicating to a virtual COM port that is implemented via USB. It may happen that the USB device is disconnected during a test (e.g. because power is removed). It is then necessary to reconnect to the 'new' device.*

**Parameters:**

| ***Name*** | ***Value*** | ***Unit*** | ***Description*** |
|---|---|---|---|
| Channel | 1...n | - | Specifies the COM port to be used |
| BaudRate | Any value | Baud | Baud rate at which the communications device operates .<br>Although any value can be specified, there is a chance that the actual baud rate will slightly differ from the specified one if 'non-standard' values are used.<br>Supported values depend on the PC hardware. |
| Databits (optional) | 7, **8** | - | Number of bits in the bytes transmitted and received.<br>Supported values depend on the PC hardware.<br>Default value is 8 bits. |
| Parity (optional) | EVEN, ODD, **NONE** | - | Parity scheme to be used.<br>Supported values depend on the PC hardware.<br>By default, no parity bit is used (NONE) |
| StopBits (optional) | **1**, 2 | - | Number of stop bits to be used.<br>Supported values depend on the PC hardware.<br>Default value is 1 |
| TerminalMode | NONE,<br>CR,<br>CR_RX_ANY,<br>LF,<br>LF_RX_ANY,<br>**CRLF** | - | Determines whether transmitted strings are terminated by a single CR, a single LF, both CR + LF characters or nothing at all.<br>In case CRLF is specified, RECEIVE_COM also accepts packets that have a single CR or LF character.<br>If NONE is specified, then the RECEIVE_COM command must specify the size of the expected packet.<br>Also see Table 1: EOL parameter values in section CONFIG_SERIAL |

**Example:**

```
CONFIG_COM[3] Baudrate = 19200, TerminalMode = LF;
```

## CONFIG_TCP*

```
CONFIG_TCP [ Channel ]  PORT = Port, EOL = Eol;
CONFIG_TCP [ Channel ]  PORT = Port, EOL = Eol,
                          AUTOCLOSE = <OnOff>;
CONFIG_TCP [ Channel ]  HOST       = "Host",  PORT = Port, EOL = Eol;
CONFIG_TCP [ Channel ]  HOST       = "Host",  PORT = Port, EOL = Eol,
                          AUTOCLOSE = <OnOff>;
CONFIG_TCP [ Channel ]  CLOSE;
```

*Not available in My-TEST and eC-my-test

**Description:**

Establishes a TCP connection to a server application on the specified host and port number. After having done this, it is possible to communicate with the application via dedicated transmit and receive commands (see TRANSMIT_TCP and RECEIVE_TCP).
The last version closes a previously opened port.
By default, closing of the port is automatically done at the end of the test so the CLOSE version is only needed when the port must be closed during the execution of the test. It is however also possible to keep the connection open during the rest of the test session by specifying AUTOCLOSE = OFF. In this case, the connection is not closed until the end of the session or until the CLOSE command is issued.

*Note: it is possible to use the command several times in the same script. Each time a configuration is specified (the first four command versions) the TCP port will be automatically closed and reopened with the new settings.*

**Parameters:**

| ***Name*** | ***Value*** | ***Unit*** | ***Description*** |
|---|---|---|---|
| Channel | 1...n | - | Specifies the channel to be configured. Note that the channel number is an arbitrary number that shall be specified in subsequent calls to TRANSMIT_TCP and RECEIVE_TCP. |
| Host (optional) | String | - | The name of the computer on which the server application is running. If the *HOST* parameter is not specified, then 'localhost' (127.0.0.1) is assumed.<br>It is possible to either specify the host name or a valid IPV4 or IPV6 address. If the IP address format is incorrect, the host will not be found. Note that e.g. "172.018.130.161" does not work due to the leading zero in the second part of the address |
| Port | 0.. | - | TCP port on which the server application is listening |
| Eol | NONE,<br>CR,<br>CR_RX_ANY,<br>LF,<br>LF_RX_ANY,<br>**CRLF** | - | Determines whether transmitted strings are terminated by a single CR, a single LF, both CR + LF characters or nothing at all..<br>In case CRLF is specified, RECEIVE_TCP also accepts packets that have a single CR or LF character.<br>If NONE is specified, then the RECEIVE_TCP command must specify the size of the expected packet. For the other three values, both CR and LF are interpreted as 'end of packet' delimiters when receiving packets, independently of the value specified here.<br>Also see Table 1: EOL parameter values in section CONFIG_SERIAL |
| Autoclose (optional) | **ON**, OFF, 0..n | - | If 'on', the connection will be automatically closed at the end of the test. Otherwise, the connection remains open during the entire duration of a test session or until it is explicitly closed. |

**Example:**

```
CONFIG_TCP [1] HOST = "Shuttle43", PORT = 5000, EOL = CRLF;
```

See also the RECEIVE_TCP* command.

## CONFIG_VISA*

**CONFIG_VISA [ *Channel* ] INSTRUMENT = Instrument;**
**CONFIG_VISA [ *Channel* ] INSTRUMENT = Instrument, AUTOCLOSE = <*OnOff*>;**
**CONFIG_VISA [ *Channel* ] CLOSE;**

*Not available in My-TEST and eC-my-test

**Description:**

Establishes a VISA connection to the specified VISA instrument. After having done this, it is possible to communicate with the VISA instrument via dedicated transmit and receive commands (see TRANSMIT_VISA and RECEIVE_VISA).
The last variant of CONFIG_VISA closes a previously opened port.
By default, closing of the connection is automatically done at the end of the test so the CLOSE version is only needed when the connection must be closed during the execution of the test. It is however also possible to keep the connection open during the rest of the test session by specifying AUTOCLOSE = OFF. In this case, the connection is not closed until the end of the session or until the CLOSE command is issued.

*Note: it is possible to use the command several times in the same script. Each time a configuration is specified (the first two command versions) the connection to the current VISA instrument will be automatically closed and reopened with the new settings.*

**Some notes on VISA:**

VISA is a standard I/O language for instrumentation programming. VISA by itself does not provide instrumentation programming capability. VISA is capable of controlling VXI, GPIB, or Serial instruments and makes the appropriate driver calls depending on the type of instrument being used. When debugging VISA problems it is important to keep in mind that this hierarchy exists. An apparent VISA problem could in reality be the results of a bug or installation problem with one of the drivers into which VISA is calling.

One of VISA's advantages is that it uses many of the same operations to communicate with instruments regardless of the interface type. For example, the VISA command to write an ASCII string to a message-based instrument is the same whether the instrument is Serial, GPIB, or VXI. Thus, VISA provides interface independence. This can make it easy to switch interfaces and also gives users who must program instruments for different interfaces a single language they can learn.

The commands presented in this manual use the industry-wide VISA API to communicate with the instruments. TEST-OK does nor provide an implementation of VISA itself. In order to make the VISA commands work, it is therefore necessary to install a VISA Runtime package. All major test equipment manufacturers provide freely downloadable installations.

**Parameters:**

| ***Name*** | ***Value*** | ***Unit*** | ***Description*** |
|---|---|---|---|
| Channel | 1...n | - | Specifies the channel to be configured. Note that the channel number is an arbitrary number that shall be specified in subsequent calls to TRANSMIT_VISA and RECEIVE_VISA . |
| Instrument | String | - | The name of the VISA instrument. As an example, this can be something like ”USB0::0x0699::0x0341::C010570::0::INSTR” or a more readable alias like “FunctionGenerator_1” if such an alias is assigned to the instrument (using an external application). |
| Autoclose | **ON**, OFF, 0..n | - | If 'on', the connection will be automatically closed at the end of the |

| *Name* | *Value* | *Unit* | *Description* |
|---|---|---|---|
| (optional) | | | test. Otherwise, the connection remains open during the entire duration of a test session or until it is explicitly closed. |

**Example:**

```
CONFIG_VISA [ 1 ] INSTRUMENT = "USB0::0x0699::0x0341::C010570::0";
```

See also the RECEIVE_VISA*command.

# User Interaction

## ASK

**ASK "*Message*", TYPE = *type* ;**
**ASK "*Message*", TYPE = *type,* TIMEOUT = *Timeout*;**
**ASK "*Message*", TYPE = *type,* TIMEOUT = *Timeout ELSE ConditionMode* ;**
**ASK "*Message*", TYPE = *type*, PICTURE = *picture* ;**
**ASK "*Message*", TYPE = *type*, PICTURE = *picture,* TIMEOUT = *Timeout* ;**
**ASK "*Message*", TYPE = *type*, PICTURE = *picture,* TIMEOUT = *Timeout ELSE ConditionMode* ;**

**Description:**

Opens a dialog box of type <type> and waits for response of the user. Execution of the test is suspended until the user response.
If <type> is OK, then only an OK button is showed, if <type> is YESNO, then a 'Yes' and a 'No' button are showed and if <type> is INPUT, then an OK button is shown together with an input field in which the user can enter a value.
If a <picture> is specified, then the dialog box will show this picture. This can be useful to indicate an area of the UUT to the user for e.g. visual inspection.

When specifying a <timeout>, the dialog will close automatically after the specified period has elapsed and the built-in #_ERROR_ variable will be set to 1 (see also #_ERROR_ : Handling errors programmatically). By default, the test is aborted and no subsequent tests are executed. If this is not desirable then a different <condition mode> can be specified and a test on #_ERROR_ may be used to decide what to do next.
If <type> is INPUT, then any key press in the input box will restart the timeout period.

The result when the user closes the dialog box is considered as an input, just like the result of any of the TEST_xxx commands. This means that the #_IN_# internal variable is set (see the section about value substitution in messages) and that it is also possible to assign the result to a variable.
The value of the result depends on the TYPE of the ASK command:

- OK
  The resulting value is always 1
- YESNO
  If the user answered YES, then the resulting value will be 1, otherwise the resulting value will be 0. Note that the keywords 'YES' and 'NO' are defined as 1 and 0 and can be used in any expression. This wildcard can only be used at the end of the string.
- INPUT
  The value entered by the user.

**Parameters:**

| ***Name*** | ***Value*** | ***Unit*** | ***Description*** |
|---|---|---|---|
| Message | String | - | Message to be presented to the user |
| Type | OK, YESNO, INPUT | - | Type of dialog |
| Picture | Filename (string) | - | Filename of the Bitmap or JPEG picture to show as part of the dialog box. If no path is specified, the file is assumed to be in the script directory of the current test suite. |
| Timeout | 0..n | ms | Timeout period. If nothing is received withing the specified time, the command will fail. |

| *Name* | *Value* | *Unit* | *Description* |
|---|---|---|---|
| | | | Specifying 0 means the command will never timeout. |
| ConditionMode | IGNORE, CONTINUE , ABORT, **ABORT_ALL** | - | If a timeout occurs, then the condition mode indicates whether the test shall be aborted immediately or whether subsequent tests shall still be executed.<br>By default ConditionMode = ABORT_ALL so if this parameter is omitted the current test is aborted and subsequent tests will be NOT executed. |

**Examples:**

The 'OK' form of the command is typically used to let the user do some action that cannot be done by the test fixture itself. When the user has finished the requested action, the user clicks on OK and the test will continue.

```
ASK "Move the jumper to position 2-3", TYPE = OK;
```

*Note that this command is also very useful during debugging of a test. It allows to temporarily suspend the test so that one can manually check e.g. some voltages on the UUT.*

The next example shows how to use the answer of the user to decide whether the test fails or not.

```
VAR #Reply;

#Reply = ASK "Does the back light function correctly", TYPE = YESNO;

IF  (#Reply == NO )
    FAIL "Problem with back light";
ELSE
    //  Do some other stuff
ENDIF;
```

In case there is communication with an external COM port whose number may change, it is possible to ask the user for the current port number and use that in the script:

```
VAR #Com;

#Com = ASK "Please enter COM port number", TYPE = INPUT;

CONFIG_COM [#Com] BAUDRATE = 9600, EOL = CRLF;
```

If the user response shall be given within a certain period, the timeout parameter can be used. In the example below, the ASK dialog is closed after 10 seconds and we then decide programmatically what to do next (the condition mode is explcitly set to CONTINUE. If this parameter is omitted, the test session will be aborted)

```
VAR #Com;

ASK "Press enter when you have placed jumper JP2", TYPE = OK, TIMEOUT =
10000 ELSE continue;
IF ( #_ERROR_ == 1 )
    SET_DIGITAL[ GROUP 101, BIT 3] = OFF;     // prepare for testing without
the jumper
ENDIF;

.... any other test code comes here
```

## SHOW_MESSAGE

**SHOW_MESSAGE "*Message*" ;**
**SHOW_MESSAGE "*Message*", PICTURE = <*filename*> ;**

**Description:**

Opens a message box without any buttons. Unlike the ASK command, execution of the test is not suspended and the next command will be executed immediately.
If a <picture> is specified, then the dialog box will show this picture. This can be useful to indicate an area of the UUT to the user for e.g. visual inspection.

The message box will be closed when the HIDE_MESSAGE command is executed or when the end of the test is reached.
If called again before closing first, the existing message box will be updated with the next text and, optionally, the new image. There will never be more than one message box showed.

***Avoid calling this more than a few times per second (e.g. in a loop). Otherwise the system may become slow and unresponsive (the user would not be able to read the different messages anyway...)***

**Parameters:**

| *Name* | *Value* | *Unit* | *Description* |
|---|---|---|---|
| Message | String | - | Message to be presented to the user |
| Picture | Filename | - | Filename of the Bitmap or JPEG picture to show as part of the dialog box. If no path is specified, the file is assumed to be in the script directory of the current test suite. |

**Examples:**

During a test, the user shall activate a switch on the UUT. It is of course possible to use an ASK command for this but that would require the user to click on the 'OK' button.
If the test script can detect when the switch is activated, there is no need for the user to indicate the event. By using a SHOW_MESSAGE command, the test will ask the user to toggle the switch and then starts monitoring the switch. As soon as the switch has toggled, the test can close the message box with the HIDE_MESSAGE command and can continue the remainder of the test.

```
//--- Ask the user to switch the switch and wait until the switch input is
on
SHOW_MESSAGE "Toggle switch S1";
WAITWHILE ( 20000 )
    TEST_DIGITAL [ 1 ] EXPECT ( == OFF );
//--- OK, switch is on. Remove message and continue the test
HIDE_MESSAGE;

..... test ......
```

The fact that the scrip execution continues also allows for other actions to be executed while waiting for the user like e.g. blinking a LED. This is showed in the example below:

```
MAP $LED ON DIGITAL OUT GROUP 1, BIT 6;

Var #Switch   = OFF;
Var #LedState = ON;
```

```
//--- Ask the user to switch the switch and wait until the switch input is
on
SHOW_MESSAGE "Toggle switch S1";

WHILE ( #Switch == OFF )

    SET_DIGITAL[$LED] = #LedState;
    #LedState = (#LedState + 1) & 1;
    WAITMS 300;
    #Switch = TEST_DIGITAL [ 1 ];

ENDWHILE;

//--- OK, switch is on. Remove message and continue the test
HIDE_MESSAGE;

..... test ......
```

*Note the use of 'OFF' and 'ON' which are equivalent to 0 and 1 when used in conjunction with digital inputs and outputs.*

See also the TEST_TIME command which allows to end the WHILE loop above after a certain period of time.

## HIDE_MESSAGE

| **HIDE_MESSAGE;** |
|---|

**Description:**

Closes a previously opened message box (See SHOW_MESSAGE). If no message box was displayed, then the command has no effect.

**Parameters:**

none

**Examples:**

See SHOW_MESSAGE command.

## PLAY_SOUND

**PLAY_SOUND FILENAME = "*File*";**
**PLAY_SOUND FILENAME = "*File*", MODE = *Mode*;**
**PLAY_SOUND STOP;**

**Description:**

Plays the specified sound file. Depending on MODE, the command will wait until the sound file is completely played or not. If not waiting, the playing of the file can be stopped by using the third variant of the command: 'PLAY_SOUND STOP'.

The command searches the following directories for sound files: the script directory; the Windows directory; the Windows system directory; directories listed in the PATH environment variable; and the list of directories mapped in a network.
The command will fail if the specified sound file cannot be found.

The sound file must be playable by an installed waveform-audio device driver.

**Parameters:**

| ***Name*** | ***Value*** | ***Unit*** | ***Description*** |
|---|---|---|---|
| FileName | String | - | The sound file to be played. |
| Mode (Optional) | **WAIT**, NOWAIT | - | If WAIT, the command will wait until the complete file has been played. When specifying NOWAIT, the command will finish immediately. |

**Example:**

Play a file from subdirectory 'Sounds' in the script directory and wait until the playing has terminated:

```
PLAY_SOUND FILENAME = "Sounds\Cow.wav", MODE = WAIT;
```

Start playing a sound file and wait for a digital input to become '1' before it is stopped (it will stop as well if the sound file is shorter then the time it took for the input to become '1'):

```
PLAY_SOUND FILENAME = "D:\TestPatterns\1Khz.wav", MODE = NOWAIT;
WAITWHILE ( 2000 ) "No sound detected"
    TEST_DIGITAL [ 14 ] EXPECT ( != 1 );  // 'wait while not 1'
PLAY_SOUND STOP;
```

## Miscellaneous

### LOG

**LOG "*Message*";**
**LOG "*Message*", INDENT = *Value*;**
**LOG "*Message*", RECORD = *RecordMode*;**
**LOG "*Message*", INDENT = *Value, RECORD* = *RecordMode*;**

**Description:**

Shows the specified message type on the log screen. The message is printed after the specified number of white spaces (zero if INDENT is omitted).

When the <record mode> parameter is ON, then the message will not be displayed on the screen but is stored in the TEST-TRACK database as part of the test results for the script in which it is executed. Note that this option is not available in My-TEST and eC-my-test.

**Parameters:**

| ***Name*** | ***Value*** | ***Unit*** | ***Description*** |
|---|---|---|---|
| Message | String | - | |
| Indent (Optional) | Integer | Number of characters | |
| RecordMode (Optional) | ON, **OFF** or 0...n | - | If ON, the message will be recorded as part of the test results in the TEST-OK database if TEST-TRACK is used and will not be printed in the log screen<br>Default record mode is OFF when no record mode is specified.<br>See also STORE_VALUE |

**Example:**

```
VAR #MyVar;

#MyVar = TEST_ANALOG[2];

LOG "=== This is a small demo for the LOG command ===",INDENT = 0;
LOG "Analog input 2 is #MyVar# mV",                    INDENT = 4;
SET_DIGITAL [ GROUP 1, BIT 3 ] = 0;
LOG "Output set to #_OUT_#",                             INDENT =
4;
```

Running this script will result in the following text in the log window:

```
[Info   ] === This is a small demo for the LOG command ===
[Info   ]    Analog input 2 is 9266 mV
[Info   ]    Output set to 0
```

## STORE_VALUE*

**STORE_VALUE KEY = *key*, VALUE = *value***

*Not available in My-TEST and eC-my-test

**Description:**

Stores a key / value pair (KVP) into the database in association with the test session that is currently running. The 'key' is a unique identified for some item of data.
If more than one command for the same KVP is used in the same script or other scripts of the test session, then the older data of the key is overwritten. Unlike the RECORD = ON option of the LOG command, the KVP is associated with the test session and not the individual test.

When running a test session (in TEST-TRACK) without a serial number, then the command is ignored and nothing is stored.

**Parameters:**

| *Name* | *Value* | *Unit* | *Description* |
|---|---|---|---|
| Key | String | - | The key that identifies the values to be stored |
| Value | String | - | Type of dialog |

**Examples:**

During a test, the idle current and the voltage of a voltage regulator on the UUT are measured and need to be stored for later reference. Each value will be stored under a different name (the 'key') so that they can be easily distinguished later on.

```
VAR #IdleCurrent;
VAR #V_reg;

#IdleCurrent = TEST_ANALOG[3];
#V_reg = TEST_ANALOG[5];

STORE_VALUE KEY = "Iidle", VALUE = #IdleCurrent;
STORE_VALUE KEY = "Vreg", VALUE = #V_reg;
```

.

## VAR

**VAR #*VariableName*;**
**VAR #*VariableName* = *Expression*;**
**VAR GLOBAL #*VariableName*;**
**VAR GLOBAL #*VariableName* = *Expression*;**

**VAR #*VariableName[]*;**
**VAR #*VariableName[]* = { *ListOfValues* };**
**VAR GLOBAL #*VariableName[]*;**
**VAR GLOBAL #*VariableName[]* = { *ListOfValues* };**

**Description:**

Defines a variable for use in the remaining part of the script or test suite. There is no need to put the declaration at the beginning of the script, as long as the declaration is made before the first use of the variable.
Without using the GLOBAL specifier, the variable is defined for the remainder of the test and will be forgotten. However if a variable is defined GLOBAL, then the variable remains valid for the entire test suite and will keep its last set value during the rest of test suite. The variable will also remain valid during the repeat of one or more failed tests.
Besides single variables, it is possible to define array variables by add '[]' behind the name. The first element in the array is at index 0.

A variable may be defined with or without explicit initialization. The first form simply defines a variable name and implicitly initializes its value to 0 or “0” depending on whether the variable is used in an integer of string context. The second form will initialize the variable's value to the explicitly specified value (which can be a simple constant or an expression). Initialization of arrays is done by specifying one or more values or expressions in a comma separated list within a pair of { } brackets (also see section Arrays).
GLOBAL variables will only be initialized the first time a declaration is found.

Variable names always start with a '#' character, followed by one or more upper/lower case letters, numbers and underscores ('_'). Other characters are not allowed.
Names are case sensitive, i.e. #MyVar and #myvar are two separate variables.

See also the general description of variables in section Variables.

**Parameters:**

| ***Name*** | ***Value*** | ***Unit*** | ***Description*** |
|---|---|---|---|
| VariableName | String | - | The name of the new variable. |
| Expression | Any valid expression resulting in an integer | - | An initial integer value for the variable. |

**Example:**

In this example one variable, #Corrected, is global which means it can be used in subsequent scripts in the test suite. This e.g. may be useful when scripts rely on previously measured values

```
VAR #Input;
VAR #Offset = 23;
VAR GLOBAL #Corrected;
VAR #Expected = #Offset * 10;  // use just defined variable for
initialization

#Input = TEST_ANALOG[1];

#Corrected = #Input + #Offset;

IF ( #Corrected < 100 )
   SET_ANALOG[2] = 4.5;
ENDIF;
```

More examples can be found in section Variables

## RUN

**RUN “CommandLine”;**
**RUN “CommandLine” ELSE *ConditionMode*;**
**RUN “CommandLine” EXPECT *Condition*;**
**RUN “CommandLine” EXPECT *Condition* ELSE *ConditionMode*;**

**RUN “CommandLine”, HIDE = *Hide*;**
**RUN “CommandLine”, HIDE = *Hide* ELSE *ConditionMode*;**
**RUN “CommandLine”, HIDE = *Hide* EXPECT *Condition*;**
**RUN “CommandLine”, HIDE = *Hide* EXPECT *Condition* ELSE *ConditionMode*;**

**RUN “CommandLine”, MODE = *Mode*;**
**RUN “CommandLine”, MODE = *Mode* ELSE *ConditionMode*;**
**RUN “CommandLine”, MODE = *Mode* EXPECT *Condition*;**
**RUN “CommandLine”, MODE = *Mode* EXPECT *Condition* ELSE *ConditionMode*;**

**RUN “CommandLine”, MODE = *Mode*, HIDE = *Hide*;**
**RUN “CommandLine”, MODE = *Mode*, HIDE = *Hide* ELSE *ConditionMode*;**
**RUN “CommandLine”, MODE = *Mode*, HIDE = *Hide* EXPECT *Condition*;**
**RUN “CommandLine”, MODE = *Mode*, HIDE = *Hide* EXPECT *Condition* ELSE *ConditionMode*;**

**Description:**

Executes the specified <CommandLine> as an external program in a separate process. The execution of the remainder of the script will be suspended until the externally started program has terminated.
Variable substitution in the "CommandLine", as described in section Values in output strings, is also supported. This makes it possible to execute programs with variable parameters (e.g. passing the serial number of the UUT to the executed program).
In TEST-TRACK, the working directory of the executed program is the script directory of the currently loaded test suite.
The exit value of the externally started program can be retrieved by assignment to a variable. If no 'EXPECT' condition is specified then the RUN command is considered to fail if the exit value is other than 0, otherwise it is the EXPECT condition that determines whether the RUN command succeeds or fails.

There are two main modes: in the first (HIDE = OFF), the application will start in a separate window and all user interaction is displayed in this window. This applies e.g. to application with a graphical interface. For console type applications, it is possible to run them 'hidden' (HIDE = ON): no external 'DOS box' is opened but instead all output of the console is redirected to the log screen. In this mode, it is also possible to pass keystrokes to the application.

***To run a 'DOS' command or a batch file (or e.g. a Visual Basic script), you must start the command interpreter:***

```
RUN "cmd.exe /c dir", HIDE = ON;   // show a directory listing in the log window

RUN "cmd.exe /c <batch file>";     // run a batch file


Also note that it may be necessary to omit the '.bat' extension on Windows
7 systems in order to correctly execute the batch file
```

***Executing a shortcut file is also possible but beware of the fact that Windows invisibly adds the extension '.lnk' to the file. Without specifying the extension it will not work.***

***It is not possible to run an external program directly from a network drive (e.g. a path starting with '***

***\'). In this case, it is necessary to 'map' the network drive to a local drive letter.***

**Parameters:**

| *Name* | *Value* | *Unit* | *Description* |
|---|---|---|---|
| Commandline | String | - | The command line must contain the name of the program to be executed, optionally followed by one or more options. The first white-space-delimited token of the command line specifies the program name. If a long file name is used that contains a space, use quoted strings to indicate where the file name ends and the arguments begin. If the file name does not contain an extension, .exe is appended. Therefore, if the file name extension is .com, this parameter must include the .com extension. If the file name ends in a period (.) with no extension, or if the file name contains a path, .exe is not appended. If the file name does not contain a directory path, the system searches for the executable file in the following sequence:<br>1. The script directory.<br>2. The 32-bit Windows system directory.<br>3. The 16-bit Windows system directory.<br>4. The Windows directory.<br>5. The directories that are listed in the PATH environment variable. |
| Mode (Optional) | **WAIT**, NOWAIT | - | If WAIT, the command will wait until the Commandline has finished. When specifying NOWAIT, the command will not wait for completion. In case of NOWAIT, the HIDE parameter is ignored and is explicitly set to OFF. |
| Hide (Optional) | ON, **OFF**, 0 ..n | - | Determines whether the program is started with its own window or without. Any integer value other than 0 will be interpreted as 'ON' |
| Condition (Optional) | | - | If no 'EXPECT' condition is specified then the RUN command is considered to fail if the exit value is other than 0, otherwise it is the EXPECT condition that determines whether the RUN command succeeds or fails. See section Single Operand Condition (in 'EXPECT' parameter) for its format. |
| ConditionMode (Optional) | IGNORE, **CONTINUE** , ABORT, ABORT_ALL | - | If the condition fails, then the condition mode indicates whether the test shall be aborted immediately or whether subsequent tests shall still be executed. By default ConditionMode = CONTINUE so if this parameter is omitted subsequent tests will be executed. (also see section Testing) |

**Examples:**

1) Opening an external text editor

```
VAR #ReturnValue;

#ReturnValue = RUN "notepad test.txt";
```

Running this script will result in notepad to show up with a file called 'test.txt'. When terminating the editor

via Exit menu, the command finishes successfully. If however the editor is forced to terminate with e.g. task manager while the 'save' dialog is open, the RUN command will fail.

2) Running an application with spaces in its path

Directory and file names in Windows may contain spaces. In order to keep the different parts of the name together, it is necessary to add double quotes (") around the entire path. Since double quotes are already used by the RUN commons to surround the entire command, it is necessary to indicate that the double quotes around the path are part of the RUN command's string. This can be done by 'escaping' them.
The example below also uses backslashes (\') in the file path name. Since the backslash is the 'escape' character, a second backslash shall be added in order to have a backslash in the final string. Alternatively, the backslashes can be replaced with a slash '/'.
See also section Special characters in strings: escape codes

```
RUN " \"C:\\Program Files\\Atmel\\AVR Tools\\STK500\\STK500\\" -cUSB -
dATmega88PA -if#_HEX_FILE_# -ieHexFiles\\eeprom.hex -I250000 -e -pb -vb -
EF9 -GF9 -fDFCC -FDFCC" \"    " ELSE ABORT ALL;
```

3) Showing the contents of a text file in the log window

Some externally run programs may have output log or error information to a file. Using a second RUN command it is possible to display the contents of such a file in the log window. In the example below, the name of the file is set in a variable. This is useful when the same file (with a possibly very long path name) is used in several places.
Note the (escaped) quotes around the string that is assigned to the variable, these are necessary to pass a path that contains spaces as a single argument to 'type':

```
VAR #ErrorFile = " \"C:\\MyApp\\Test Results\\Error.log\" ";
VAR #ReturnCode;

#ReturnCode = RUN " \"C:\\MyApp\\DoIt.exe\" ";
IF ( #ReturnCode != 0 )
   RUN "cmd.exe /c type #ErrorFile# ", HIDE = ON;
ENDIF;
```

4) Running a visual basic script file:

```
RUN "cmd.exe /c MyScript.vbs";
```

## DISABLE_BOARDDETECT

***DISABLE_BOARDDETECT;***

**Description:**

Disables the detection of UUT removal during test. Normally, a running test session is aborted immediately when the UUT is removed. However, there are situations in which the board shall be temporarily removed during the test to e.g. solder some calibration pads.
The Board Detect function will be disabled either until an ENABLE_BOARDDETECT command is encountered or until the end of the script. At the start of each script, the Board Detect function is implicitly enabled.

**Parameters:**

None

**Example:**

```
... some code to determine whether Resistor R1 must be placed or
not ...

DISABLE_BOARDDETECT;
ASK “Remove the board, solder R1 and place the board again”, Type =
OK;
ENABLE_BOARDDETECT;
```

The user can safely remove the board from the fixture without the abortion of the test suite.

## ENABLE_BOARDDETECT

***ENABLE_BOARDDETECT*;**

**Description:**

Enables the detection of UUT removal during test. This may be necessary after disabling the function with a previous DISABLE_BOARDDETECT command.
See the DISABLE_BOARDDETECT for more information and an example

**Parameters:**

None

## REPAIR*

**REPAIR "*Problem*", CLASS = *Class*;**
**REPAIR "*Problem*", CLASS = *Class*, *ConditionMode*;**

*Not available in My-TEST and eC-my-test

**Description:**
This command makes the test fail like the FAIL command. Additionally, the specified problem is assigned to the UUT in the TEST-TRACK database.
If the combination 'Problem description / problem class' does not yet exist in the database then the problem is first added. The problem is not re-assigned to the UUT if the same (unrepaired) problem is already assigned to the UUT.

Whether the remainder of the test script is executed or not depends on the specified condition mode. If no mode is specified, then execution will continue.

**Parameters:**

| ***Name*** | ***Value*** | ***Unit*** | ***Description*** |
|---|---|---|---|
| Description | String | - | The problem corresponding to this description and the class (see next parameters) is added to the UUT |
| Class | VISUAL, ELECTRICAL, MIXED | | The problem corresponding to the mentioned above and the class (see next parameters) is added to the UUT |
| ConditionMode<br>(Optional) | IGNORE, **CONTINUE**, ABORT, ABORT_ALL | - | If the condition fails, then the condition mode indicates whether the test shall be aborted immediately or whether subsequent tests shall still be executed.<br>By default ConditionMode = CONTINUE so if this parameter is omitted subsequent tests will be executed. |

**Example:**

**This example asks the user to visually check the presence of a capacitor. If the capacitor is missing, the test will fail and a repair item is added to the TEST-TRACK database.**

```
Var #Reply;

#Reply = ASK "Is C2 correctly placed?", TYPE = YESNO;

if ( #Reply == NO )
    REPAIR "Capacitor C2 is missing", CLASS = VISUAL, ABORT;
endif;
```

## Conditions and Expression Syntax

All test commands and the IF command have a condition part. The condition of an IF command is not exactly the same as for the testing commands in that the condition of the IF command requires two operands that are compared (this is how most programming languages define conditions) while the test commands omit the first operand which is implicitly replaced by the measured value (see examples below as well).
Additionally, variables can be assigned new values via arithmetic expressions.

### Single Operand Condition (in 'EXPECT' parameter)

Single Operand Conditions have the basic form :

```
ComparisonOperator Expression
```

The `ComparisonOperator` is one of the well known operators such as 'greater than' or 'equal to' as listed in the table below:

| ***Operator*** | ***Description*** |
|---|---|
| == | Equal to |
| != | Not equal to |
| > | Greater than |
| >= | Greater than or equal to |
| < | Less than |
| <= | Less than or equal to |

The `Expression` can be any arithmetic expression that results in a value (see the subsection about expressions below).
For any comparison to work, it is necessary to have to operands while there is only one operand (the expression) in what is showed here. The explanation is that the Single Operand Condition is only used in TEST_xxx commands and the value read from the associated input is used as the implicit first Operand.

Example: TEST_ANALOG [2] expect < 300;

> The Single Operand Condition is " < 300". while the left hand operator is the voltage read on analog input 2. Therefore, this test will fail if the measured voltage on analog input 2 is greater or equal to 300 mV.

More complex conditions are possible as well when multiple Single Operand Conditions are combined by using boolean algebra. An example is:

> TEST_ANALOG [2] expect ( >= 300 AND <= 500)

> This test will fail if the measured voltage on analog input 2 is less than 200 mV or greater than 300 mV.

## Dual Operand Condition

Dual Operand Conditions have the form :

```
Expression ComparisonOperator Expression
```

This is much the same as the Single Operand Condition but now with the left hand operator explicitly written. This form is used in IF statements.
The same comparison operators as listed in the previous paragraph can be used.

## Expressions and Operators

An Expression is an arithmetic expression that results in a single value when evaluated. Expressions are used in the two condition types described above but they are also used to assign a value to a variable:

```
#Variable = Expression;
```

Arithmetic expressions are combinations of arithmetic operators (+, -, * etc..) and operands. Operands can be constants, variables or function calls.

Supported operators are listed in the table below. The examples following the table illustrate some of the operators.

| ***Operator*** | ***Description*** |
|---|---|
| + | Addition |
| - | Subtraction |
| * | Multiplication |
| / | Division, the result is the integer part of the division |
| % | Modulo |
| & | Bitwise AND |
| \| | Bitwise OR |
| ^ | Bitwise eXclusive OR |
| ~ | Bitwise inversion ('NOT') |
| >> | Bitwise shift right |
| << | Bitwise shift left |

**Examples**

Shift a variable one place to the left:

```
#i = #i << 1;
```

## Formal Syntax Description

### TEST_xxx Commands

```
TestCondition         :     SingleOperandComparison
                      |     ( TestCondition AND TestCondition )
                      |     ( TestCondition OR  TestCondition )
                      |       NOT TestCondition
                      |     ( TestCondition )

SingleOperandComparison     : ComparisonOperator ValueExpression
```

## IF Command

```
IfCondition             : DualOperandExpression
                        | '(' IfCondition AND IfCondition ')'
                        | '(' IfCondition OR  IfCondition ')'
                        |                 NOT IfCondition
                        | '(' IfCondition ')'

DualOperandComparison   : Expression ComparisonOperator Expression
```

## Comparisons

```
ComparisonOperator    |     =<    (TRUE if left operand is less than the
right operand)
                      |     >     (TRUE if left operand is greater than the
right operand)
                      |     <=    (TRUE if left operand is less than or
equal to the right operand)
                      |     >=    (TRUE if left operand is greater than or
equal to the right operand)
                      |     ==    (TRUE if left operand is equal to the
right operand)
                      |     !=    (TRUE if left operand is not equal to the
right operand)
```

*Note: for Single Operand Comparisons, 'left operand' is the implicit value read from the input by a TEST_xxx command.*

## Any Value

Values are used in e.g. the assignment to a variable but also in the conditions for TEST_xxx and IF commands

```
Expression            : Expression '+'  Expression
                        | Expression '-'  Expression
                        | Expression '*'  Expression
                        | Expression '/'  Expression
                        | Expression '%'  Expression
                        | Expression '|'  Expression
                        | Expression '&'  Expression
                        | Expression '^'  Expression
                        | Expression '<<'  Expression
                        | Expression '>>'  Expression
                        | '~'  Expression
                        | '-'  Expression
                        | INT  Expression
                        | STRING  Expression
                        | <integer>

                        | <floating point>
                        | <string>
                        | ON
                        | OFF
                        | VarName
                        | VarName '[' Exression ']'
                        | VarName.BITFIELD
                        | VarName '[' Exression ']'.BITFIELD
                        | '(' Expression ')'
                        | FunctionName '(' ListOfParams ')'
```

## Testing Command Examples

Note that the test commands optionally also have a condition mode and a message string, which is omitted here for clearness

**Example:**

**Comparison:**
TEST_ANALOG [$Name] EXPECT ( < 12.5 );

**Condition BooleanOperator Condition:**
TEST_ANALOG [$Name] EXPECT ( > #Min AND < #Max );

**NOT Condition:**
TEST_ANALOG [$Name] EXPECT NOT( < 12.5 );

## IF Command Examples

The operands of a comparison can be either a constant value or a variable. Examples of variables are the loop iterator of a FOR loop (the IF command shall be part of the commands that are in the FOR loop!) or names of fields in a serial packet defined in the Packet definition file (see chapter Serial communications and packet definitions)

**Example:**

**Comparison:**

```
FOR #i { 1 .. 10 }
    IF ( #i < 5 )
        SET_ANALOG [1] = 10.0;
    ELSE
      SET_ANALOG [1] = 5.0;
    ENDIF;
ENDFOR;
```

**Condition BooleanOperator Condition:**

```
WAIT_RX 100;

IF ( (#MyVar == 0b1001 ) OR ( # MyVar == 0b1100 ) )
    SET_ANALOG [1] = 10.0;
ENDIF;
```

## Serial communications and packet definitions

The TEST-OK Test Engine provides two ways to control the communication between tester and UUT over a serial channel on the Test Controller Card.
The first way of communication assumes that the UUT is able to receive one fixed-format test-packet to which it answers with another fixed-format test packet. The structure of both packets must be defined by a specification in a text file. The rest of this chapter uses the term 'Single Packet Mode' for this mode.
One of the advantages is that the test firmware in the UUT can be kept simple and that controlling and testing the UUT is done via simple SET and TEST commands similar to the commands used for digital and analog I/O.

The second way of communication allows to send and receive various packets to and from the UUT in a terminal-like way using ASCII characters. Sending is done by specifying a string which may contain variable names that will be substituted by their value.
Receiving is done be specifying what is expected. If the received packet does not match the expected then the test fails. As for transmission, the 'expected' string may contain variable place holders.
The term for this communication mode used in the remaining of this chapter is 'Terminal Mode' and also applies to other commands that allow to communicate with the UUT via a COM port or via TCP.
One of the advantages of this mode is that the number of packets and their contents is much more flexible. However it lacks the simplicity in the resulting scripts that comes with the 'Single Packet Mode'

## Single Packet Mode

Commands exist to set fields (or bits in a field) in the packet to be transmitted to the UUT and to test on fields (or bits in a field) in received packets. The packet structure can be defined in a script file, using a specific syntax.
It is also possible to define a checksum field in either transmitted and received packets and how to calculate them.
If a checksum field is defined for the transmitted packet, the checksum will be automatically calculated and filled into the packet before every transmission. If a checksum is defined for the received packet, then the checksum is calculated over every received packet and checked against the checksum field in the packet. Only if the checksum is correct, the packet will be available to the running test (i.e. When a WAITRX command is executed, then the next command will not be executed until a packet with a correct checksum is received or until the specified timeout period has expired).

### Transmission of packets

Packets can be transmitted in two modes: automatic or manual (set with the CONFIG_SERIAL command). In automatic mode, a packet will be sent about every 100ms, independent of whether the test has set any field or not. In manual mode, a packet is only sent for every TRANSMIT_SERIAL command encountered in the test.

When manual mode is used, it is possible to synchronize with the transmission of packets by using the WAITTX command. The command after the WAITTX command will only be executed after a packet is sent. The WAITRX command can be used to synchronize on the reception of a packet as explained above.

Using the SET_SERIALFIELD and TEST_SERIALFIELD commands, it is possible to set new values in the fields of the transmitted packet and to read and test values from both the received packets and the transmitted packet. See the description of these two commands for more information.

## Definition of the packets

This section describes the format of the text file that defines the transmitted and received packets. After the explanation of the various parts, an example file is shown and explained.

A script file defining the transmitted and received packets shall always start with the text:

*[packetdefinitions]*

in order to distinguish is from a regular test script.
Next will be an indication that defines whether multi-byte fields in the packet are big-endian or little-endian, i.e whether the first byte of a multi-byte field is the most significant byte or the least significant byte:

*Endianess = MSBYTE_FIRST;*

or

*Endianess = LSBYTE_FIRST;*

Next follow two sections, one for the transmit packet and one for the receive packet. Each section starts with

*[TX]*

or

*[RX]*

The structure of both sections is identical: a number of lines, all terminated with a semicolon (;), that define a field. The order in which the fields are defined is the order in which the fields appear in the packet.
Each field definition looks like this:

*$<Fieldname> : Size = <Size>, Default = <DefaultValue> ;*

The <Fieldname> part defines the name of the field. Similar to mapping names in the test scripts, there must always be a '$' in front of the name (so that there will never be confusion between any user defined name and keywords of the test language.
<Size> is the size of the fields in bytes. The minimum value is 1 byte.
<DefaultValue> is the value that the field will have at the start of every test. This avoids starting every test with a number of commands to set the fields of the transmit packet. It also may avoid incorrect interpretation of the receive packet before any packet is received. For Rx packets, the specification of a default value is optional.

There is one special format for the default value which is used to define a checksum field. When the keyword 'Calc' occurs in the default value (see the exact format below), then the field is considered as a checksum field which is automatically calculated and filled in for transmitted packets and which is checked on correctness for received packets.
The definition of a checksum field looks like this:

*$Checksum : Size = <Size>, Default = CALC( <Range>, <Method>, <StartValue> ;*

(note that the name of this field may be any name, just like the other fields).
The <Range> indicates the bytes in the packet over which the checksum shall be calculated. The first byte of the packet is numbered '1'. If the checksum is to be calculated over the first 8 bytes in the packet, the the <Range> shall be specified as :

1 .. 8

The <Method> field indicates the calculation method. Possible choices are 'XOR', 'ADD' and 'SUB'.
The XOR method calculates the checksum be exclusive-ORing all packet bytes in the specified range on a byte-by-byte basis. The ADD method also works on bytes but simply adds all bytes together while the SUB method subtracts all bytes from the start value. Note that the checksum field may be up to 4 bytes large.

The <StartValue> specifies the initial value when calculating the checksum.

*Note that is it theoretically not necessary for the checksum to be the last field in the packet. But in practice it is.*

## Definition example

```
////////////////////////////////////////////////////////////////////////////
////////////////////////////
// This file contains the description of the packets that will be
transmitted and received from the UUT.
////////////////////////////////////////////////////////////////////////////
////////////////////////////

[PacketDefinitions]

// Endianess values:
//   MSBYTE_FIRST
//   LSBYTE_FIRST
Endianess = MSBYTE_FIRST;

[TX]
//--- TestRequest packet

$PacketId:       Size = 1, Default = 0x55;
$RtcHour:        Size = 1, Default = 0;      // The hour to be set to the
RTC
$RtcMinutes:     Size = 1, Default = 0;      // The minutes to be set to
the RTC
$MiscControl:    Size = 1, Default = 0;      // SetRtc and Buzzer on/off
$SdControl:      Size = 1, Default = 0x13;
$LcdControl:     Size = 1, Default = 0;
$Checksum:       Size = 1, Default = CALC( 1..6, XOR, 0x5A );

[RX]

//--- Test Reply packet
$PacketId:       Size = 1, Default = 0xAA;
$RtcHour:        Size = 1, Default = 0;     // The hour read from the RTC
$RtcMinutes:     Size = 1, Default = 0;     // The minutes read from the
RTC
$RtcSeconds:     Size = 1, Default = 0;     // The seconds read from the
RTC
$RtcStatus:      Size = 1, Default = 0;     // RTC Control Register 2
contents
$SdStatus:       Size = 1, Default = 0;     // SD interface inputs
$Keys:           Size = 1, Default = 0;     // State of the four key inputs
$LeftCount:      Size = 1, Default = 0;     // Left clicks counter
$RightCount:     Size = 1, Default = 0;     // Right clicks counter
$SwVersion:      Size = 2, Default = 0;     // Version of the UUT firmware
$HwRevision:     Size = 1, Default = 0;     // Revision of the UUT hardware
$Checksum:       Size = 1, Default = CALC( 1..12, XOR, 0x5A );
```

## STX Mode explained

As mentioned in the section about the CONFIG_SERIAL command, it is possible to enable 'STX Mode' when using the Single Packet Mode.
Sending packets in STX mode is a way synchronize the receiver with the stream of bytes that is received. The principle (and implementation) is simple: each packet starts with a 'Start of Transmission' (STX) byte which has a unique value that will never be transmitted in the actual packet data. The end of a packet is marked with another unique value: the 'End of Transmission' byte (ETX). Synchronizing is now very easy for the receiver: as soon as an STX byte is received, the start of the packet is found and as soon as ETX is received, the packet has been completely received. Besides the synchronization, another advantage is that the receiving software does not need to have any knowledge about the packet size.
If the packet contains bytes with the same value as STX and ETX, then these values cannot be transmitted as such since this would make the receiver resynchronize on the start of a new packet or on the premature end of the current packet. Instead, a third unique value is defined to 'escape' the normal meaning of the special values. The DLE (Data Link Escape) byte is therefore inserted into the transmission on every occurrence of STX, ETX and DLE in the packet data. Thus, when receiving DLE, then this byte shall not be added to the packet but the next received byte shall be added without interpreting the received value as STX, ETX or DLE.

Values used by the TCC1800:
STX = 0x0F ($15_{10}$)
ETX = 0x04 ($4_{10}$)
DLE = 0x05 ($5_{10}$)

## Script Example

Using the packet definition from the previous sub section, it is possible to write a script that will test the Real Time Clock (RTC) on the UUT. For this purpose, the transmitted packet contains a field called $MiscControl containing a bit (bit 0) called 'SetRtc'. If this bit is set, the firmware in the UUT will copy the value of the field $RtcHour and $RtcMinutes into the RTC and the seconds register will be reset.
At the same time, each reply packet sent by the UUT contains the current value of the hours, minutes and seconds register of the RTC. Bit 4 of the field '$RtcStatus' is a status bit from the RTC that is 1 if the RTC is in error mode.
Note that the RTC stores the time in BCD format.
It is now possible to test the RTC by first writing a new time into the RTC, wait e.g. three seconds and then checking the reported RTC contents in the reply packet:

```
CONFIG_SERIAL [ 1 ] BAUDRATE = 38400, PARITY = NONE,
                    TXMODE = AUTO, RXMODE = AUTO,
                    RXTHRESHOLD = 32, RXTIMEOUT = 10,
                RTXMODE = ON;

SET_SERIALFIELD[1] $RtcHour     = 0x12;
SET_SERIALFIELD[1] $RtcMinutes  = 0x30;
SET_SERIALFIELD[1] $MiscControl = 1;

//--- Wait until the request has been sent.
WAITTX [1] 500; //  wait until the request has been sent

//--- Now wait until we got answer back. If not doing this, we may look at an 'old'
packet in which the new time is not yet reported
WAITRX [1] 500;

//--- Verify that the new values are accepted by the RTC and can be read back.
//--- Also check that the RTC's status bit is 0.
TEST_SERIALFIELD [1] [$RtcHour]     EXPECT ( == 0x12 ), "Error on Hour (#_IN_#)";
TEST_SERIALFIELD [1] [$RtcMinutes]  EXPECT ( == 0x30 ), "Error on Minutes (#_IN_#)";
TEST_SERIALFIELD [1] [$RtcSeconds]  EXPECT ( == 0 ),    "Error on Seconds (#_IN_#)";
TEST_SERIALFIELD [1] [$RtcStatus.4] EXPECT ( == 0 ),    "Error on RtcStatus. Reading
#_IN_#";

//--- Now wait three seconds and check whether the seconds counter is running
WAITMS 3000;

TEST_SERIALFIELD [1] [$RtcSeconds]  EXPECT ( > 1 ),    "RTC is not running (#_IN_#)";
```

*Note that in this example, we used the automatic send/receive mode (see CONFIG_SERIAL command) and StxMode was 'on'.*

The automatic send mode has the advantage that the script is not cluttered with explicit TRANSMIT_SERIAL commands while automatic received mode is mandatory in the Single Packet Mode.

## Terminal Mode

*Note: the features described in this section also applies to the COM port, TCP and VISA communication commands (TRANSMIT_COM, TRANSMIT_TCP, TRANSMIT_VISA, RECEIVE_COM, RECEIVE_TCP and RECEIVE_VISA). Their configuration commands are slightly different however, see the appropriate sub sections in section* Configuration.

The term 'target' is used to indicate the device to communicate with. In case of TRANSMIT_SERIAL this is the UUT but in case of the similar COM, TCP and VISA commands this can be any device (e.g. a remote controller multimeter or a software application).

### Configuration

When using this mode, it is not necessary to specify a Packet Definition File. Instead ASCII strings are sent and transmitted, each string or 'packet' being terminated with a <CR> and/or <LF> character.

Using the CONFIG_SERIAL command, the baud rate and other parameters are set as for the Single Packet Mode with a few exceptions:

- TxMode and RxMode shall be set to 'Manual'
- StxMode shall be set to 'off'
- Unless all packets to be received have the same length, RxThreshold shall be set to a large value
- RxTimeout shall be set to a period that is as short as possible to improve reactivity but it should be longer than the interval between individual bytes in a packet (normally, there will be no time in between bytes). A good indication e.g. the time it takes to send 2 bytes
- Eol indicates whether each packet is terminated with an End Of Line character(s).
  In ASCII Terminal Mode each packet is usually ended with a Carriage Return (CR) and/or Line Feed (LF) character. It is however possible to suppress the EOL and transmit and send packets with only the bytes specified in the TRANSMIT_SERIAL and RECEIVE_SERIAL commands

For more details, see the description of the CONFIG_SERIAL command.

## Packet Transmission

The message specified in the TRANSMIT_xx command is send over the channel associated with the command. There are mainly two different types of communication:

- pure ASCII communication
- communication with any kind of packet

Pure ASCII communication corresponds best with the title of this section: 'terminal mode'. In this type of communication, each packet ('message') is followed by a CR and/or LF to indicate the end of the message. Answers are terminated with CR and/or LF as well.
Values of variables can be sent as a string of ASCII characters by using variable placeholders (see section Values in Output Strings).

**Examples**:

```
VAR #Result = 145;
TRANSMIT_SERIAL[1] "Hello World";
TRANSMIT_SERIAL[1] "The result is #Result# (which is #Result:2x# in hex)";
```

The two messages that are sent are:
"Hello world"  and "The result is 145 (which is 91 in hex)", both terminated with CR, LF or CR+LF depending on the EOL parameter of the last CONFIG_SERIAL command (not shown here)

The second type of communication is much more flexible. Besides ASCII characters, it is possible to send bytes any of any value, not just ASCII. Targets using this kind of packets usually do not use EOL characters to indicate the end of the packets and to achieve this, the EOL parameter of the CONFIG_TCP, CONFIG_COM of CONFIG_SERIAL command can be set to NONE.
The values to be sent can be specified in two ways: either as constants in ASCII or 'escaped hex' notation (section 2.4.3) or as variable placeholders (section Values in Output Strings). It should be noted that the variable placeholders allow to send the value as a binary single byte or multi-byte value, which is different from the example above in which the value 145 was sent as three ASCII characters '1', '4' and '5'.  Look at:

```
CONFIG_TCP [1] HOST = “Shuttle43“, PORT = 5000, EOL = NONE;
TRANSMIT_TCP[1] "\x02#Result:r2m#\x03";
```

The ':r2m' format specifier after the variable name means that the value is sent as a 'raw' value in 2 bytes with the most significant byte first.
A total of four bytes is sent with hexadecimal values of:  02 00 91 03.

## Packet Reception

As for packet transmission, terminal-like ASCII communication or more advanced communication is possible.

Characters received from the target are stored in a FIFO buffer and each time a RECEIVE_xxx command is executed, the TestEngine will check whether there is at least one EOL character (CR and or LF) in the FIFO. If so, then all characters up to the first EOL character are read. If not, then then command will wait until an EOL character is received or a timeout occurs.
If EOL was configured as NONE, then reading from the FIFO will end as soon as the expected number of bytes is read (see the SIZE parameter of the RECEIVE_xx commands in the Testing section).

Each RECEIVE_xx command specifies the bytes /characters that are expected. This may be a literal string of characters, like:

```
RECEIVE_SERIAL[1] "Hello world";
```

If only part of the string is known (because the received string may contain a part that is variable, e.g indicating the current date) then it is possible to use wildcards. A wildcard is a special symbol that represents one or a variable number of bytes or characters.
As an example, if the target shall answer with something like "Hello, current date is 10-03-1985. Self test OK", but with any date, then the receive command may look like this:

```
RECEIVE_COM[1] "Hello, current date is *. Self test OK";
```

The '*' will match zero or more characters until the input contains the character just following the '*', i.e. a period ('.') in the case above. For a more specific check we may choose to specify:

```
RECEIVE_COM[1] "Hello, current date is ??-??-????. Self test OK";
```

The '?' will match exactly one character. If the exact value of the date must be checked, one can use variable placeholders. In the case above, the day, month and year values are ASCII coded decimal values and the command becomes:

```
VAR #Day;
VAR #Month;
VAR #Year;
RECEIVE_COM[1] "Hello, current date is #Day#-#Month#-#Year#. Self test OK";
```

If the received string is "Hello, current date is 10-03-1985. Self test OK" the #Day will become 10, #Month will be 3 and #Year will be 1985. Using IF etc. then allows to check whether the values are as expected.

When packets are non-ASCII, they are often organized in 'fields' and do not use EOL characters (EOL = NONE). As an example, a packet that is transmitted by the target consists of seven bytes: the first byte is always 3, followed by two 16-bit values (coded LSByte first), a one byte checksum and terminated with a fixed value byte of 15. If we're interested in the two field values but not in the checksum, then we may write this:

```
VAR #Field1;
VAR #Field2;

RECEIVE_COM[1] "\x03#Field1:r2l##Field1:r2l#?\x0F", SIZE = 7;
```

Both #Field variable's format specifiers are 'r2l' indicating raw, 2-byte LSByte-first values. Together with the preceding '\x03' byte, this matches the first 5 received bytes. The '?' wildcard will match one byte, and the last received byte must by 0F (hexadecimal presentation of 15).

As a last example, consider the we are just happy if we receive a 10-byte packet as long as it starts with 'S' and ends with a byte with value 6:

```
RECEIVE_COM[1] "E*\x06", SIZE = 10;
```

Additional information on the strings that can be sent and how received strings can be processed is described in the sections about the TRANSMIT_SERIAL and RECEIVE_SERIAL commands.