TESTFRAME ENGINE

VERSION 2008.06

USER'S GUIDE

TestFrame Engine User's Guide

If you have any comments or suggestions regarding this document, please send them via e-mail to ccctesting@logica.com

For more information on Logica and TestFrame visit:
http://www.logica.com
http://www.testframe.com

For more information on the TestFrame Engine write to:
ccctesting@logica.com

Central Competence Centre Testing
Telephone: 020-5033000
Fax: 020-5033011

TABLE OF CONTENTS

WELCOME TO THE TESTFRAME ENGINE

Welcome to the TestFrame Engine, Logica's tool for processing automated software tests designed using the TestFrame methodology. The Engine is the heart of TestFrame's automated test environments, linking together test specifications, test software, and test results.

Using this guide

This guide describes how to use the TestFrame Engine to specify, automate, and process your automated software tests.

The first chapter offers an introduction to TestFrame and its automated test environments.

The second chapter gives an overview of the TestFrame Language (TFL) with which to write test specifications.

Chapter three and four describe test reports and the Engine Message Centre.

The last three chapters comprise a reference manual for the TestFrame Engine; they give full descriptions of its settings, built-in action words, and interface functions.

1 INTRODUCTION

1.1 TestFrame

TestFrame is Logica's methodology for structuring test processes. An important characteristic of TestFrame is the emphasis it places on the separation between *what* is to be tested, and *how* it is to be tested. This separation lies at the core of TestFrame. It is carried through in the entire realization phase, splitting it effectively into two separate phases: the *analysis* and the *navigation* phase. Both have their own design and implementation steps.

Analysis concerns itself with *what* is to be tested. It can consist of several activities: analyzing the *system under test* (SUT); defining which functions of the SUT are to be tested, and creating test conditions and cases for these; breaking the test cases down into test actions; writing the test actions down in cluster files as *action words*, using a specified notation called *TestFrame Language* (TFL).

Navigation deals with *how* is to be tested. This phase only exists when tests are automated. It is software development of project specific test products. It can consist of several activities: setting up an environment for automated testing of the SUT, making a technical analysis of the SUT and providing its necessary interfaces; designing a structure for the test software; implementing the test actions defined during the analysis phase in *action word functions*.

Making the separation between analysis and navigation has several advantages. Firstly, the deliverables of both phases can be *developed* separately. This makes it possible for several people to work on a test without getting their wires crossed. Secondly, the deliverables can be *maintained* separately. So, for instance, a change in a test case does not necessarily mean a change in an interface with the SUT, or vice versa. Thirdly, creating tests is made more *flexible*—new tests can reuse the deliverables of the old ones more easily. A new test case, for instance, might be constructed out of previously defined test actions; if these already have implementations, it need not even be implemented anew, only in the cluster. Or if, for example, a system is ported to a new operating system; most test cases will still hold, while only their corresponding action word functions must be replaced.

Separating test data from interfacing to the SUT means that for test *execution*, these two must be brought together. This is done by using a piece of software called the *Engine*. It is the link between analysis, navigation and execution. The Engine must be called upon by the test software to get data from the cluster and to log the results, so a test report can be generated.

1.2 Automated test environments

The figure below sketches an abstract picture of an automated test environment in which TestFrame is applied.

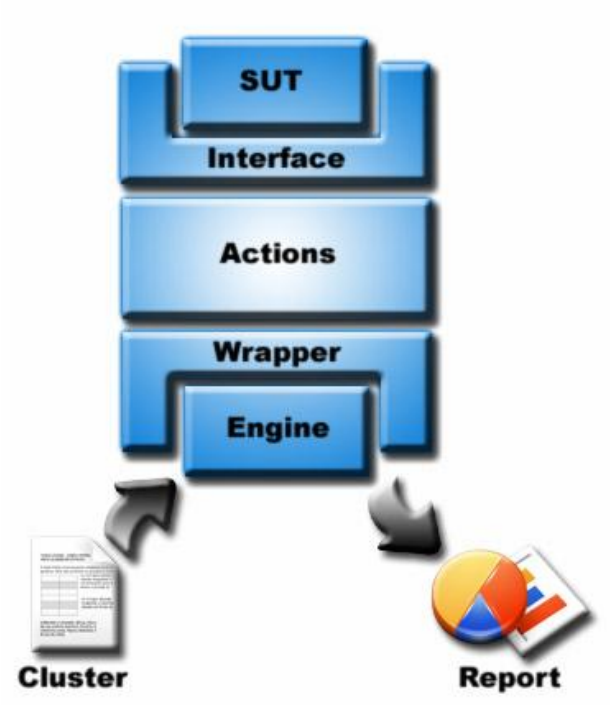

Figure 1.1

An automated test environment using TestFrame, in seven parts:

1. SUT, the system under test;
2. CLUSTER, a tab separated file specifying the actions of the test, written in the action word based TestFrame Language;
3. REPORT, a document containing the results of the test;
4. ACTIONS, the test specific software automating the test actions;
5. ENGINE, the TestFrame Engine linking together cluster, report and test software;
6. WRAPPER, the test tool specific interface to the Engine;
7. INTERFACES, the interfaces to the SUT used by the test software.

TestFrame's basic premise—the separation between analysis and navigation—grew out of the desire to overcome the limitations of record-and-playback tools. These tools were, and still are, used to create test software. They contain standard interfaces to certain types of applications—most of them mainly to GUI applications. They can record actions performed on the SUT—generating scripts with the used interfaces and data. These scripts can then be run to replay the actions. But if an action differs only slightly from another one—e.g. typing "B" instead of "A" in some field of an administrative SUT—a whole new test script is required for playback by these tools.

To improve this situation, TestFrame takes the test data out of the scripts and places it in other files—the clusters; furthermore, it places the test results in files outside the test software as well—the test reports. By doing so, specification and implementation have been separated.

To be able to let test tools—or more general: test software—get test data from clusters and write results to reports, there must be software that is able to do this. This software is the *Engine*, the user's guide to which you are now reading. The functionality of the Engine can be used by the test software

via interface functions. Not only by scripts in various test tools, but by almost any piece of software. So, on any SUT to which interfaces exist, automated tests using TestFrame can be performed. (Since any system which does not have interfaces isn't of use to anyone, it can be said: all SUTs.) Because, if no tools exist which support the SUT's interfaces, software can be developed which does. This interfacing software—written in C, Java or whatever language—can implement the necessary test actions and interface with the Engine via its interface functions to process and report the tests.

The interface functions of the Engine are the same in any environment and on any platform. This creates the added advantage that all test automation with TestFrame uses the same standard. A project testing on a Windows XP PC a GUI application with Quick Test Pro has as its basis the same structure as a project testing an embedded system via a parallel cable from a UNIX workstation using software written in C.

In the test environment model it is clear that besides separating the Engine, the remaining test software is divided into three parts: the interfaces to the SUT; the actions, which comprise the project specific test software—the implementations of the user defined action words; and a third module, the wrapper.

The wrapper contains the generic part of the test software which deals with test processing. It is a test software language specific "shell" around the Engine (hence the name wrapper). Its purpose is to offer the Engine's functions to the actions module, and be the Engine's active counterpart. The Engine has been built as a passive program—primarily because of most test tools' property to take complete control of the machine on which they run. This control means the initiative must be taken by software in the tools. So the wrapper takes these initiaves: calling on the Engine to process the cluster until it finds a test action, then calling on the actions module to execute this action, then returning to the Engine again.

At this moment wrappers are available for some of the most well known test tools: Quick Test Pro® by HP™, QARun™ by Compuware®, Rational Robot™ by IBM®, WinRunner by HP and also for the programming languages, Visual Basic, C/C++ and Java. Wrappers for SilkTest(Borland), TestComplete(Automated QA) and Rational Functional Tester(IBM) are in development.

When installing the TF Engine, wrappers for WinRunner, QARun, C/C++, Java and Visual Basic are installed by default. Other wrappers can be requested by sending an email to ccctesting@logica.com.

Traditional GUI based information systems will most likely still be tested using one of the standard test tools. In those cases, the three modules of test software will all be part of this tool: the interfaces to the SUT provided by the tool's manufacturer, the actions and wrapper as scripts written in the tool's scripting language.

Technical systems—basically all non-GUI applications—will be tested using special tailored software, since test tools offer no solutions. The interfaces—in technical systems more diverse and specific—the actions and the wrapper will all be programmed individually—in the same, or different programming languages.

2 TESTFRAME LANGUAGE: AN OVERVIEW

2.1 Introduction

This chapter gives an overview of the syntax and semantics of TFL—the TestFrame Language. TFL is a language in which tests designed via the TestFrame methodology can be written. So TestFrame is the method and TFL is the notation system for this method. For tests to be processed correctly by the Engine, they have to be written in TFL. Files containing tests written in TFL will be referred to as TestFrame clusters or test clusters or—simply—clusters. In this chapter about TFL it will be explained how to write clusters.

2.1.1 File format of clusters

The TestFrame Engine expects clusters to be written in a specified format—a tab delimited text file. In this format the file is divided into lines, separated from each other by a newline character. Each line in its turn is divided into units—referred to as columns or arguments—separated by tabs.

One can of course use all types of programs to create tab delimited text files—spreadsheet programs or professional text editors.

2.2 Basic TFL features

2.2.1 Action words

An action word is a unique identifier that represents a specific action that is to be performed. There can be only one action word on a cluster line, and this action word has to be in a specific argument of that line—the first one. The exception to this is when the Engine operates in Master mode, then the action word should be in the second argument.

2.2.2 User defined action words

A user defined action word is an action word that indicates that the test executioner must perform a specific action. In case of automated testing this means some piece of software associated with this action word is to be executed—this is called the action word function.

2.2.3 Built-in action words

A built-in action word is an action word that indicates that the Engine itself has to perform an action. A complete list of all built-in action words can be found in chapter 6.

2.2.4 Action word parameters

Action words can be parameterized. These parameters are placed in arguments after the action word. By using parameters, actions that are similar but use different data, can be indicated by the same action word with different parameter values.

2.2.5 Test lines

A test line comprises the cluster lines which contain an action word and its parameters—including the tab and newline characters.

2.3 Improving cluster readability

2.3.1 Empty lines

Empty lines can be added to a cluster—before or after any one or any group of test lines, as many as desired, and distributed throughout the cluster as desired. In terms of characters, an empty line is no more than an extra newline character in the cluster file.

2.3.2 Comment lines

TFL offers the possibility to add comment lines to a cluster. For comment lines the action word column must be empty; except for Master/Slave configurations, this is the first column.

2.3.3 Argument descriptions

Comment lines can contain descriptions of the action word arguments. The Engine offers functionality to store and retrieve these argument descriptions. It should be noted that this functionality is limited. Since the argument descriptions are not explicitely registered to a specific action word, determining them is a matter of interpretation for the Engine. To avoid misinterpretation, the comment line containing the desired argument descriptions should be the last comment line before the line(s) whose corresponding action word function(s) call these descriptions.

2.3.4 Splitting test lines

TFL offers the possibility to split a test line over several cluster lines. To do this, the continue token must be used. The continue token can be set to any value by adding it to the ini file containing the Engine's settings—to the key *ContText* in the section *SYSTEM*. The default value for the continue token, however, is **&Cont**.

The continue token must be placed in the argument following the last one of each broken-off cluster line and in the first argument of each continued cluster line.

Example 2.1

The following cluster demonstrates the continue token.

	1	2	3	5
1	**check names**	John	Sue	&Cont
2	&Cont	Betty	Steve	&Cont

3	&Cont	Angela		

Note that this test line has six arguments—the first being the action word itself—and the argument with index number 4 (the third parameter of the action word) contains the value *Betty*.

2.4 Cluster header

The cluster header is a collection of test lines—not comment lines—at the beginning of a cluster. These test lines contain action words of a specific class: header words.

2.4.1 Header words

A header word is an action word that instructs the Engine to place a specific cluster property into the header of the test report. There are two types of header words: built-in header words and user defined header word.

2.4.2 Built-in header words

The TestFrame Engine has six types of built-in header words. Each of them is meant for a specific cluster property. The following table lists all of their names in alphabetical order, their associated cluster properties and their corresponding report tags.

name	*cluster property*	*report tag*
author	the name of the author of the cluster	Cluster author
cluster	the name of the cluster	Cluster name
date	the date of the cluster	Cluster date
document	the name of the document in which the cluster is stored	Document
sheet	the name of the spreadsheet on which the cluster is written	Sheet
version	the version of the cluster	Cluster version

Table 2.1 Built-in header words

The Engine has, besides the above mentioned English header word names, also synonyms of the built-in header words—and all other built-ins for that matter—in the three other supported languages: German, French and Dutch. These synonyms, extended descriptions and examples of the built-in header words can be found in chapter 6.

2.4.3 User defined header words

The six before mentioned built-in header words might not be sufficient to convey all information desired in a specific test project. It is therefore possible to add user defined header words to the Engine during a test run. The way to add them to the Engine is to register them using the interface function **tfe_RegisterHeaderWord**.

2.5 Cluster structuring

2.5.1 Structure elements

A structure element is a group of test lines which are marked as belonging together. TFL supports four different structure elements, i.e. four types of marks: test case, test condition, scenario and section. In TFL only the beginning of a structure element can be marked. The beginning of the next instance of this structure element type automatically marks the end of the previous one. The scope of a structure element is the cluster file; i.e., sub clusters or templates called from within a structure have structure elements of their own.

2.5.2 Built-in action words for structuring

As all other statements in TFL, marking the start of a structure element in a cluster must be done by using an action word: a built-in action word for structuring. Table 2.2 lists all four built-in action words for structuring clusters. For these action words there are also synonyms in German, French and Dutch. These synonyms, extended descriptions and examples of these built-in action words can be found in chapter 6.

action word	*associated structure element*
test case	test case
test condition	test condition
scenario	Scenario
section	Section

Table 2.2 Built-in action words for structuring

2.6 Cluster variables and commands in arguments

2.6.1 Literals and commands

There are two types of arguments: literal arguments and arguments with commands. Literal arguments are arguments whose content is used as-is. The Engine makes no changes and does no processing, but simply takes as value the string entered in the cluster.

The Engine regards arguments by default as literals. Only if they start with one of the command prefixes does the Engine perform a specific operation on the argument.

In Table 2.3 the eight supported argument commands are listed alphabetically, together with their default prefix and a description of the command that the Engine is given when this prefix is used.

command name	*default prefix*	*description of the command*
Anything command	&Anything	stating explicitly an argument with arbitrary value
Date command	&Date( )	determination of relative dates in specified formats
DateTime command	&DateTime( )	determination of relative date and time in specified formats
Empty command	&Empty	stating explicitly an empty argument
Expression command	&	evaluation of an arithmetical or string expression (including references to cluster variables)
Keep command	&Keep( )	assigment of a cluster variable value from an outside application
Not-empty command	&NotEmpty	stating explicitly a filled argument
Snap command	&Snap( )	saving data from an outside application to file in order to create a baseline for regression testing
Spaces command	&Spaces( )	setting the argument value to a string consisting of a specified number of spaces

Table 2.3 Argument commands

2.6.2 Cluster variables

It is possible in TFL to make use of cluster variables. Cluster variables are variables just as in programming languages or mathematical equations—keywords in which data can be stored and to which can be referred by name.

The characters that can be used in the variable names are limited—only letters, digits and underscores are allowed. Another condition posed upon a variable name is that its first character must be either a letter or an underscore. Note that this means that variable names can not contain spaces. Once again, remember that also cluster variables are case sensitive.

When referring to a cluster variable, this argument must start with the expression prefix. Cluster variables are created the first time data is assigned to them; this will be considered their declaration. This means that assignment of data to a cluster variable must be the first act performed on them, and this must be done before they are ever referred to.

The lifetime and scope of a cluster variable is dependent upon the place it is declared, since these variables are file bound. In effect, this means that as long as the cluster in which it is declared is not finished, the cluster variable exists, and as long as this cluster is the one that is currently being processed, the cluster variable is in scope.

2.6.3 Expressions

The expressions TFL allows are stated in Table 2.4.

operation	operator	valid usage	return type
sub-expression	( )	(EXPRESSION)	EXPRESSION
unary plus	+	+ NUMBER	NUMBER
unary minus	-	- NUMBER	NUMBER
multiplication	*	INTEGER * INTEGER	INTEGER
		INTEGER * FLOAT	FLOAT
		FLOAT * INTEGER	FLOAT
		FLOAT * FLOAT	FLOAT
division	/	INTEGER / INTEGER	INTEGER if left-operand is multiple of right-operand; otherwise FLOAT
		INTEGER / FLOAT	FLOAT
		FLOAT / INTEGER	FLOAT
		FLOAT / FLOAT	FLOAT
concatenation	+	STRING + STRING	STRING
		STRING + NUMBER	
		NUMBER + STRING	
addition	+	INTEGER + INTEGER	INTEGER
		INTEGER + FLOAT	FLOAT
		FLOAT + INTEGER	FLOAT
		FLOAT + FLOAT	FLOAT
subtraction	-	INTEGER – INTEGER	INTEGER
		INTEGER – FLOAT	FLOAT
		FLOAT – INTEGER	FLOAT
		FLOAT – FLOAT	FLOAT
assignment	=	VARIABLE = EXPRESSION	EXPRESSION

Table 2.4 Summary of expression operations

The following table lists the operators in order of precedence; those with higher precedence are placed above those with lower precedence; those of equal precedence are placed on the same line. The associativity of the operators is also listed.

expression type	operators	associativity
primary expression	( )	left-to-right
unary expression	+ −	right-to-left
multiplicative expression	* /	left-to-right
concatenation/ additive expression	+ −	left-to-right
assignment expression	=	left-to-right

The following example illustrates various expressions.

Example 2.2

The following cluster illustrates several different operators; for the setting precision is taken: 4g.

	1	2	3
1	**cluster**	Expressions example	
2			
3		*expression*	*equivalent value*
4	**display sum**	&12 + 30	42
5	**display difference**	&53 - 11	42
6	**display product**	&6.1 * 11	67.1
7	**display quotient**	&7 / 18	0.3889
8	**display quotient**	&12/ 3	4
9	**display value**	&-2	-2
10	**display value**	&+4.8	4.8
11	**display value**	&12 + 9.34 * 7.1	78.31
12	**display value**	&Var = 12	12
13	**display value**	&(-6.2 + 13) / (-2 * Var)	0.2833
14	**display string**	&"hello " + "world"	hello world
15	**display string**	&2001 + ": A Space Odyssey"	2001: A Space Odyssey

The value of any argument is converted to a string; also if the argument contains an expression which is evaluated to a number. Conversion of integers to strings is straightforward; floats to strings, however, require some explaining. This float-to-string conversion is done as specified by a user defined setting **Precision**; this setting specifies how to handle exponential/floating point notation, and how many digits to take; see paragraph 5.9 for the explanation on how to use **Precision**.

2.6.4 Exporting and importing variables

The method TFL offers to transfer data from one cluster file to another is by exporting and importing cluster variables. With both actions built-in action words are associated—**export variable** and **import variable** respectively. Chapter 6 contains full descriptions of the action words, including the synonyms in German, French and Dutch.

The built-in **export variable** must be used to transfer data out of a cluster file. When this action word is processed, the names of the variables that are given as arguments, are stored together with their values—their values at that particular moment in the cluster, that is. Not only is the exported name-value pair stored in memory, but it is also written to file: the keep file. In this keep file—an ini file—it is added to the section **[EXPORT]** with the name of the variable as a key and the value of the variable at the moment of export as the value corresponding with this key.

To transfer data into a cluster file the built-in action word **import variable** must be used. When **import variable** is processed, the specified variables are taken from the keep file and added to the list of variables of that cluster, and given the value it was assigned in the keep file.

2.6.5 Keep command

The method to get data *from an application* and place it—runtime—into the cluster is called the keep method. In a parameter of a user defined action word must be placed the keep command with as its parameter a variable in which the data from outside is to be stored. In the corresponding action word function—the action word's implementation in an application—the interface function **tfe_KeepArgument** or **tfe_KeepParameter** must be called. With this function the index of the argument or parameter that should contain the keep command is given, as is, of course, the value that is to be kept. When the Engine receives the **tfe_KeepArgument** or **tfe_KeepParameter** call, it

checks if the specifies argument or parameter does indeed contain the keep command, and if so, assigns the kept data to the cluster variable.

To denote in a cluster that an argument is a keep command, it must start with the keep prefix. For a keep command the parameter can only be the name of a cluster variable. In this cluster variable the kept value is going to be stored; note that this variable need not be declared before, since a (successful) keep command is also considered an assignment, and, consequently, a declaration.

Example 2.3

In the following cluster demonstrates how the keep command is used in a cluster.

	1	2	3	4
1	**cluster**	Keep example		
3				
4		*name*	*address*	
5	**get name and address**	&Keep(Name)	&Keep(Address)	
6				
7		*name*	*address*	
8	**send letter**	&Name	&Address	

When a keep is performed, the kept value is not only stored internally—as the variable's current value—but it is also written to file. This is also the keep file, only not to the section **[EXPORT]**, but to the section **[KEEP]**. As with exported variables, it is stored as a key-value pair with the name of the variable as key. Although kept values are written to the keep file, this does not mean that they can be imported. Export/import and keep are different operations. So, if it is required that data that comes from an external application, should also be used in other clusters later on, this data must be explicitly exported—just as all other data in a cluster.

2.6.6 Snap command

For creating a baseline in regression testing, TFL contains the snap command. A cluster for a regression test can be made, but in the arguments where the baseline data is to be placed, the snap command can be entered. When this "prototype" cluster is run—with the same action word functions as if it were an actual regression test—instead of system data being compared with argument values, this data is written to file.

As with the keep command, data from an outside application is used when a snap is performed. So, this data must as well be entered into the Engine via an interface function in the corresponding action word function. The interface function in this case is either **tfe_CheckArgument** or **tfe_CheckParameter**. As function parameters are given a string and an index of an argument/action word parameter. With this argument value the specified string is to be compared. If the argument contains the snap command, instead of comparing, the specified string is snapped—i.e. written to the snap file.

To state that an argument contains the snap command, it must start with the snap prefix—**&Snap**. A snap command may have a parameter—i.e. *may*—it is allowed to have a parameter, but also to have no parameter. The difference lies in the way data is written to the snap file. If a parameter is used, this parameter is then taken by the Engine as the key to which the data will be written in the snap file. If no parameter is specified in the snap command, the Engine generates the key to which the data will be written in the snap file. The Engine makes this key unique by using properties of the argument in which the snap command is placed: its index and the number of the test line to which it belongs. The fourth argument of the test line numbered twelve will be given as snap key: `(L: 12, A:4)`; argument seven of test line twenty-three: `(L: 23, A:7)`.

2.6.7 Date command

The Date command transforms the value of an argument into a date in a specified format, and relative to the current date—i.e. current at the moment of test execution. To state that an argument contains a date command, it must start with the date prefix.

The date command has four parameters—the first one is mandatory, the other three are optional. The parameters must be separated by commas. This first parameter is a string which specifies the format. The other three are integers specifying the number of days, months and years respectively, which are added to (or, if the numbers are negative, subtracted from) the current date. Table 2.5 summarizes the parameters of the date command.

#	*description*	*type*	*mandatory/optional*
1	date format	string	mandatory
2	day offset	integer	optional (default: 0)
3	month offset	integer	optional (default: 0)
4	year offset	integer	optional (default: 0)

Table 2.5 Date command parameters

If the offset parameters are omitted, a value of 0 is assumed—i.e. the current day, month or year. The date format must start and end with a double quotation mark. Within these quotation marks any character can be used—some characters, or combinations of characters are special, though, since the date command replaces these by numbers representing the (relative) dates. Table 2.6 shows a list of the special characters and character combinations and of what they represent.

character (combination)	*represents*
d	number of the day
dd	number of the day, padded with a leading zero if less than ten
m	number of the month
mm	number of the month, padded with a leading zero if less than ten
yyyy	number of the year in four digits
yy	number of the year in two digits, leaving out the century

Table 2.6 Special characters of the date command format

2.6.8 DateTime command

The DateTime command transforms the value of an argument into a date and/or time in a specified format, and relative to the current date and time—i.e. current at the moment of test execution. To state that an argument contains a datetime command, it must start with the datetime prefix.

The datetime command has eight parameters—the first one is mandatory, the other seven are optional. The parameters must be separated by commas. This first parameter is a string which specifies the format. The other seven are integers specifying the number of days, months, years, hours, minutes, seconds and milliseconds respectively, which are added to (or, if the numbers are negative, subtracted from) the current datetime. Table 2.7 summarizes the parameters of the datetime command.

#	*description*	*type*	*mandatory/optional*
1	date format	string	mandatory
2	day offset	integer	optional (default: 0)
3	month offset	integer	optional (default: 0)
4	year offset	integer	optional (default: 0)
5	hour offset	integer	optional (default: 0)
6	minutes offset	integer	optional (default: 0)
7	seconds offset	integer	optional (default: 0)
8	milliseconds offset	integer	optional (default: 0)

Table 2.7 Date command parameters

If the offset parameters are omitted, a value of 0 is assumed—i.e. the current millisecond, second, minute, hour, day, month or year. The datetime format must start and end with a double quotation mark. Within these quotation marks any character can be used—some characters, or combinations of characters are special, though, since the date command replaces these by numbers representing the (relative) date and time. Table 2.8 shows a list of the special characters and character combinations and of what they represent.

character (combination)	*represents*
n	number of milliseconds
nnn	number of milliseconds, padded with leading zeros if needed.
s	number of seconds
ss	number of seconds, padded with a leading zero if less than ten
m	number of minutes
mm	number of minutes, padded with a leading zero if less than ten
h	number of hours
hh	number of hours, padded with a leading zero if less than ten
D	number of the day
DD	number of the day, padded with a leading zero if less than ten
M	number of the month
MM	number of the month, padded with a leading zero if less than ten
YYYY	number of the year in four digits
YY	number of the year in two digits, leaving out the century

Table 2.8 Special characters of the date command format

2.6.9 Spaces command

The spaces command transforms its argument's value to a specified number of spaces. This command offers nothing different from simply typing the specified number of spaces directly as the argument. The spaces command has one parameter with which to specify with how many spaces to fill the argument.

2.6.10 Empty, NotEmpty and Anything commands

There can be situations in which a test designer does not want a one-on-one character comparison between expected and found data—data being there or not being there might suffice. For a specific test line the data in a particular argument can even be irrelevant.

To place information of this type in clusters, the empty, not-empty and anything commands—also collectively referred to as the check commands—can be used in action word parameters. For a parameter to consist of one of the check commands, it must start with one of their three command prefixes.

When **tfe_CheckArgument** or **tfe_CheckParameter** is called with an argument/parameter that contains:

(i) the empty command: the check is considered passed if the compared string is an empty string; otherwise it is considered failed.
(ii) the not-empty: if the string is a non-empty string, the result of the check is considered passed. Only if it is empty, will the result be a failure.
(iii) the anything command: this will always result in a passed check. It is to state that the check in this case is irrelevant, so any string—empty or not empty—is considered passed.

2.7 Flow of control

Up to now only the sequentially processing of clusters was mentioned; i.e., each cluster line in order of line number. TFL offers also constructions to repeat or to skip cluster lines. This flow of control in the cluster consists of selection statements (if-statements) and iteration statements (repeat- and while-loops.) Both if-statements and while-loops make use of conditions to determine to go one way or the other; the syntax of these conditions will be discussed first.

2.7.1 Conditions

There are several built-in action word used in flow of control structures—**if**, **else if**, and **while**—whose parameters are interpreted in a special way: as a condition. A condition is a group of parameters making up an expression, which will be evaluated to either *true* or *false*.

The simplest conditions are those in which the relation between two operands is evaluated; its prototype is as follows (taking the action word **if** for illustration.)

	1	2	3	4
1	**if**	left operand	relational operator	right operand

The parameter order determines which parameter value is taken as the left operand, which as the right operand, and which as the operator. Note that argument commands can be used—as value in the condition is taken the parameter value just as **tfe_GetArgument** or **tfe_GetParameter** returns.

There are six different relational operations, which are evaluated as stated below:

- 'equal to' operation
  If both parameter values can be converted to numbers, the condition is *true*, if the numerical values of the operands are the same; otherwise, the condition is *true*, if all characters of the operands match.

- 'not equal to' operation
  If both parameter values can be converted to numbers, the condition is *true*, if the numerical values of the operands are *not* the same; otherwise, the condition is *true*, if one or more characters of the operands do not match.

- 'less than' operation
  If both parameter values can be converted to numbers, the condition is *true*, if the numerical value

of the left operand is less than that of the right one; otherwise, the condition is *true*, if the string value of the left operand comes alphabetically before that of the right one.

- 'less than or equal to' operation
  If both parameter values can be converted to numbers, the condition is *true*, if the numerical value of the left operand is less than or equal to that of the right one; otherwise, the condition is *true*, if the string value of the left operand comes alphabetically before or is identical to that of the right one.

- 'greater than' operation
  If both parameter values can be converted to numbers, the condition is *true*, if the numerical value of the left operand is greater than that of the right one; otherwise, the condition is *true*, if the string value of the left operand comes alphabetically after that of the right one.

- 'greater than or equal to' operation
  If both parameter values can be converted to numbers, the condition is *true*, if the numerical value of the left operand is greater than or equal to that of the right one; otherwise, the condition is *true*, if the string value of the left operand comes alphabetically after or is identical to that of the right one.

The possible operator values for each operation are listed in the table below.

equal to	***not equal to***	***less than***	***less or equal***	***greater than***	***greater or equal***
=	!=	<	<=	>	>=
==	<>	lt	le	gt	ge
eq	ne	less than	less or equal	greater than	greater or equal
is	is not	kleiner als	kleiner oder gleich	größer als	größer oder gleich
ist	ist nicht	inférieur à	inférieur ou égal à	groesser als	groesser oder gleich
est	n'est pas	inferieur a	inferieur ou egal a	supérieur à	supérieur ou égal à
equal to	is niet	kleiner dan	kleiner of gelijk	superieur a	superieur ou egal a
ist gleich	not equal to			groter dan	groter of gelijk
égal à	ist ungleich				
egal a	différent de				
gelijk aan	different de				
	ongelijk aan				

It is possible to combine several of such relational condition to one larger condition. This must be done using logical operators. Below is illustrated how two relational condition are combined.

	1	2	3	4	5	6	7	8
1	**if**	left operand 1	relational operator 1	right operand 1	logical operator	left operand 2	relational operator 2	right operand 2

Two relational conditions can be combined in two different ways: with a *logical and*, or with a *logical or.* They are defined as follows:

- logical and
  If both relational conditions are *true*, the combination is *true*; otherwise, the combination is *false*.

- logical or
  If both relational conditions are *false*, the combination is *false*; otherwise, the combination is *true*.

The possible operator values for *and* and *or* are listed in the table below.

operation	*possible operator values*			
logical and	and	und	et	en
logical or	or	oder	ou	of

If more than two relational conditions are combined, the logical combinations are evaluated from left to right; i.e., an expression `expr1 and/or expr2 and/or expr3 and/or expr4` is always evaluated as (((`expr1 and/or expr2) and/or expr3) and/or expr4`).

2.7.2 Selection statements

Selection statements express decisions about when groups of cluster lines must be processed and when they must be skipped. They are like if-statements in programming languages: an expression is evaluated, and if it is considered *true*, the lines inside the statement are executed; furthermore, TFL if-statements also offer the possibilities of alternatives (*else*) and multi-way decision statements (*else if*.)

Let's first look at simple selection statements—those consisting of only an *if*, and no *else*. To make such a statement in a cluster two test lines must be added to the cluster, each of them with a special built-in action word. The first one is a line with the action word **if**; this action word has two functions: it marks the start of a selection statement, and it specifies the condition to evaluate; its parameters make up a condition as described in paragraph 2.7.1. The second line is that with the action word **end if**; its function is to instruct the Engine where the selection statement ends; it has no parameters.

The following (part of a) cluster is an example of a simple selection statement; lines 14 and 15 are only processed if the (numerical or string) value of the variable number is less than 5.

	1	2	3	4
13	**if**	&Number	<	5
14	**do something**	A		
15	**do something**	B		
16	**end if**			

In order to create an alternative—a group of test lines which are executed if the condition in a selection statement is *false*—the built-in action word **else** must be used; its functions are to mark the end of the *true*-group, and mark the beginning of the *false*-group (the alternative.) It has no parameters, and it must always be used on a line in-between the **if** and **end if** action words.

The following (part of a) cluster illustrates the **else** action word; if the value of the cluster variable Code is equal to the string "valid" on line 21, line 22 is processed and line 24 is skipped; if it is not "valid" line 22 is skipped and line 24 is processed.

	1	2	3	4
21	**if**	&Code	=	valid
22	**open door**			
23	**else**			
24	**give error**	invalid code		
25	**end if**			

Multi-way decision statements must be created using the built-in **else if**. This action word does several things: it marks the end of the (latest) *true*-group, it marks the beginning of its own *true*-group, and it states the expression to evaluate if the last one was considered *false*. Its parameters make up a

condition, the same as those of **if**. Several **else if** lines may be used in one selection statement; they must, however, always be used in-between **if** and **end if**.

The following example illustrates a multi-decision selection statement; note than more than one **else if** is used, as well as **else**.

	1	2	3	4
31	**if**	&Name	=	Lotte
32	**do something**	A		
33	**else if**	&Name	=	Maxine
34	**do something**	B		
35	**else if**	&Name	=	Graig
36	**do something**	C		
37	**else**			
38	**do something**	D		
39	**end if**			

Line 32 is processed and all others skipped if Name is "Lotte"; if it is not "Lotte", line 32 is skipped, and the condition on line 33 is evaluated; if this condition is *true*, i.e., if Name contains the string "Maxine", only line 34 is processed before the statement is ended. Should Name be neither "Lotte" nor "Maxime", the condition on line 35 is evaluated; should this be *true*, line 36 is processed; otherwise, line 38.

Some words about nesting: it is possible to place if-statements within other if-statements. Note, though, that any **else** is connected with the last encountered **if**, which isn't already connected to an **else**; any **end if** is connected with the last encountered **if**, which isn't already connected to an **end if**.

2.7.3 Unconditional loops

Iteration statements speficy looping; they express the repetitional processing of a group of cluster lines. There are two types of loops: unconditional and conditional ones; the first type will be discussed in this paragraph, the second type in the next one.

Unconditional loops specify beforehand the number of times a group of cluster is to be processed. To state such loops in a cluster the lines which are to be repeated, must be placed in-between lines with special built-in action words; these action words are **repeat** and **end repeat**.

The purposes of the built-in **repeat** are to mark the beginning of the loop, and to state how many times it must be repeated; its parameter must contain this number; it must be the value of a non-negative integer, or an argument expression which evaluates to such a value. The built-in **end repeat** must be used to mark the end of the unconditional loop.

The following example illustrates the use of a repeat-loop in a cluster; when this cluster is run, lines 2 and 3 are processed seven times

	1	2	3	4
1	**repeat**	7		
2	**do something**	A	B	C
3	**do something**	E	F	G
4	**end repeat**			

Note that when an expression is used in the argument of the **repeat** test line, this expression is *not* evaluated for every iteration; only the value to which it evaluates at the beginning of the loop is taken as the number of iterations. Consider the following cluster.

	1	2	3	4
1	**do something**	&Number = 5		
2	**repeat**	&Number		
3	**do something**	&Number = Number + 1		
4	**end repeat**			
5	**do something**	&Number		

In the cluster above the variable Number is set to 5, and when an unconditional loop is started on line 2, the argument value evaluates to 5. This means that line 3 is processed five times, regardless of the fact that the same variable is changed during the execution of the loop. Note also that this means that after loop, the second argument of line 5 has the value 10.

2.7.4 Conditional loops

Conditional loops repeat a group of cluster lines as long as certain condition is *true*. This condition is checked each time before the lines within the loop are processed. To state conditional loops in a cluster the lines which are to be repeated, must be placed in-between lines with special built-in action words; these action words are **while** and **end while**.

The built-in **while** marks the beginning of a conditional loop, and it states its condition; its parameters specify this condition as described in paragraph 2.7.1. The built-in **end while** marks the end of a while-loop.

The following example illustrates the use of a conditional loop in a cluster.

	1	2	3	4
37	**while**	&Counter	>	15
38	**do something**	&Counter = Counter – 1		
39	**end while**			

Consider the (part of a) cluster above; if the value of the variable Counter is greater than 15, the loop is started; otherwise, the loop is skipped. Once the loop is started, line 38 is processed as long as at the beginning of each iteration the variable Counter is (still) greater than 15.

Note that if within a condition nothing is changed with regard to the condition at the beginning of the while-loop, once the loop is started, it will never end; an infinite loop has been started, one whose condition is always *true*. Be careful to avoid such constructions.

When nesting iteration statements, note that any **end repeat** is connected with the last encountered **repeat**, which isn't already connected to an **end repeat**; and any **end while** is connected with the last encountered **while**, which isn't already connected to an **end while**.

2.8 Subclusters

Sub clusters are clusters called from within other clusters. In other words: a sub cluster is a collection of test lines (and empty lines, comment lines, etc.) in a file other than the main cluster. A cluster that is a sub cluster, can have one or several sub clusters itself. When a sub cluster is called, the calling cluster is paused and is only continued after all lines of the sub cluster are processed.

2.8.1 Calling sub clusters

A sub cluster must be explicitly called by another cluster. So, the initiative must be taken by the calling cluster—in the sub cluster nothing has to be done to make it a sub cluster. The way to call a sub cluster is by using the built-in action word **do cluster**. This action word has two parameters.

The first parameter of **do cluster** must contain the file name of the sub cluster that is to be called. The specified name in the parameter must be that of an existing file. This means it must be specified with the correct extension; also the directory must be specified. It is possible to omit the directory, though; in that case, the cluster directory as specified in the Engine's settings is assumed to be the file's path.

The second parameter of **do cluster** can contain the name of the file in which the sub cluster results are to be written. This parameter is optional—when omitted, the results are placed in the report of the calling cluster. If the results of a sub cluster should be written to a separate report, this second parameter must contain the file name of this report—including directory and extension. The directory can be omitted. In that case the report directory from the Engine's settings is taken as the report file's path.

Example 2.4

The following cluster demonstrates the calling of sub clusters.

	1	2	3
1	**cluster**	UK statistics—a subcluster example	
2	**version**	1.0	
3			
4		*subcluster*	*subcluster report*
5	**do cluster**	c:\TestFrame\Clusters\England.txt	c:\TestFrame\Report\England.rtf
6	**do cluster**	NorthernIreland.txt	NorthernIreland.rtf
7	**do cluster**	d:\SomeDir\Wales.txt	
8	**do cluster**	Scotland.txt	a:\AnotherDir\Scotland.rtf

2.9 Templates

A template is a parameterised collection of test lines that can be called with a single action word. In other words: templates apply the action word principle to action words themselves—making an action word for a collection of other action words. Templates can be used to create one's own level of abstraction in clusters. Action words with corresponding action word functions can be created on a low level, and higher level action words can be formed out of these action words via templates. Templates can call other templates themselves.

2.9.1 Defining templates

In order to use a template, it must first be created—and to create a template it must be defined in a cluster. For the definition of a template, two built-in action word are to be used: **define template** and **end template**. Chapter 6 contains full descriptions of them, including their synonyms in German, French and Dutch.

The template definition consists of three parts: a test line marking the start of the definition, one marking the end, and the body of the template—which are all the lines of the template. The test line starting the template definition must contain the action word **define template**—or one of its synonyms—and the line ending the template definition the action word **end template**. Both are mandatory in any template definition.

Besides marking the start of a template definition, the action word **define template** does something else: it declares the prototype of the template. Each template must have a unique prototype. This prototype is placed in the arguments of **define template**. The prototype of a template consists of the template's name and its parameters.

Example 2.5

The following cluster demonstrates the definition of a template.

	1	2	3	4	5
1	**define template**	build house	Length	Width	Height
2					
3		*length*	*heigth*	*thickness*	
4	**build wall**	&Length	&Height	&Thickness=0.1	
5	**build wall**	&Length	&Height	&Thickness	
6	**build wall**	&Width	&Height	&Thickness	
7	**build wall**	&Width	&Height	&Thickness	
8					
9		*length*	*width*		
10	**build roof**	&Length	&Width		
11	**end template**				

2.9.2 Template files

After a template has been successfully defined, its body—all lines in between **define template** and **end template**—are written to a template file. This template file is a tab separated text file—just like the cluster. The name of the template file is based on the name of the template itself. This file name is the template's name with spaces replaced by underscores. It is always given the extension: tpt. So a template named **slice the bread** is written to a file named *slice_the_bread.tpt*. The directory in which this template file is placed, is the template directory as specified in the Engine's settings.

2.9.3 Calling templates

A template can be used by calling it. Just like an action word. So the name of the template must be placed in the column of the action words. The values that are to be given to the template parameters must be placed in the following arguments—in the order in which the parameters are specified in the prototype.

2.9.4 Declaring previously defined templates

Since templates are written to template files, they continue to exist even after the test run is finished. Another cluster—in which a certain template is not defined, but that wishes to use it anyway—can explicitly declare the template and refer to the correct template file. For this purpose TFL offers the built-in action word **declare template**. Chapter 6 states a full description –including synonyms—of this action word.

2.10 Master/Slave configuration

A test in which the actions are distributed over several Engines, is said to use a Master/Slave configuration. In a Master/Slave configuration two types of roles exist for Engines. These roles are Master Engine and Slave Engine. A Master Engine contains the information about a certain action, while the Slave Engine performs this action. This is the basic premise of Master/Slave.

Each Engine in a Master/Slave configuration runs on its own machine. All Engines can run under different operating systems, work with different test tools, and be involved in testing different application. All this possible, as long as the computers on which the Engines run, are connected to each other in a network. The Engines communicate with each other using TCP/IP—so this is a prerequisite.

2.10.1 The Master Mode

In the settings of each Engine a mode for the Master/Slave configuration must be specified. This mode can be either the stand alone mode or the Master mode. The mode can be set in the Engine's ini file in the section **[MASTERSLAVE]** with the key **Mode**. This key can have one of two values: **Standalone** or **Master**, corresponding with the stand alone mode and the Master mode respectively.

When an Engine is to be a Master Engine during the test, its mode must be set to the Master mode. In all other cases the stand alone mode must be chosen.

2.10.2 Slave Clusters

In a Master/Slave configuration it is the Slave that initiates the connection with its Master. A Slave Engine starts processing a cluster, exactly the same as a standalone Engine. At a certain point in this cluster it is stated that the Slave must connect itself to a specified Master. From that point on, the Slave receives its test lines from its Master, until the Master disconnects the Slave. After that, the Slave cluster is processed in the same fashion as before the connection. Note that the Master engine must be running at the time that the Slave is going to connect to it.

Connecting to the Master must be done with the built-in action word: **do cluster**. In the section about sub clusters—paragraph 2.8—the workings of the **do cluster** action word were explained. These remain essentially the same for Master/Slave usage; only, when **do cluster** is used for Master/Slave purposes, there must be given a link of some kind to the Master Engine. This be done via a URL using the **TestFrame Master/Slave Protocol** (**tmsp**.) The syntax of **tmsp** is as follows:

tmsp:// *<master machine name>* **:** *<port number>* **?slavename=** *<slave name>*

Example 2.6

The following cluster is that of a Slave Engine in a configuration with one Master. Besides some header information, all the processing that is done by the Slave, it gets from its Master. This Master runs on a machine that is called **Svengali** and in its settings is specified that the port number for Master/Slave connections is **4100**. The Slave, in turn, calls itself **Trilby**.

	1	2	3
1	**cluster**	Slave example	
2	**version**	2.1	
3	**author**	Trilby	
4			
5	**do cluster**	tmsp://Svengali:4100?slavename=Trilby	TrilbysResults.rtf

2.10.3 Master clusters

A Master cluster contains test lines that are to be processed by different Engines. Some lines are meant for the Master Engine itself, others for its Slaves—each line separately for one particular Slave, though.

To specify which line is meant for which Engine, the Slave column must be used. This is the first column in Master clusters. This column is used for Master/Slave communication. In this column

must be specified which Engine is to process a particular test line. If the first argument is left empty, the line will be processed by the Master Engine itself. If it contains the name of a Slave, the line is sent to that Slave.

To make sure that a Slave is connected to a Master before this Master sends test lines to the Slave, the Master cluster can use a built-in action word **connect slave**. The arguments of **connect slave** are the names of the Slaves that must be connected to the Master Engine from this point on in the Master cluster. To avoid a deadlock: after a certain time out period the cluster will be continued—whether all specified connections are established or not. This time-out period is five minutes.

After Master and Slave have been synchronized, the Master can send test lines to the Slave. To do this, the test line for a particular Slave must be placed in the Master cluster, and in the Slave column of this line—i.e. the first argument—must be placed the name of that Slave.

A test line which is sent to a Slave, is going to be processed by that Slave Engine, not by the Master Engine. That means that all the information in that line must be known by the Slave Engine. The action word, for example, of a test line sent to a Slave, must be registered to that Slave Engine—not the Master. The same with cluster variables, templates, etc.

The Master always disconnects its Slaves when it reaches the end of the Master cluster. A Slave can be disconnected by the Master before the end of the Master cluster. This must be done explicitly using the action word **disconnect slave**. The parameters of disconnect slave are the names of the Slaves that must be disconnected from the Master Engine from this point on in the Master cluster.

The Master/Slave mechanism works *asynchronously*. This means that a Master sends a line when it encounters it in the cluster. Regardless of whether or not the Slave to which it is sent, has finished its previous lines. Each Slave buffers its lines it is sent, and processes them when it's ready—but in the order the Master specified. So, it is possible that the Master cluster has been completely processed, while the Slaves are still processing test lines—each Slave at their own pace.

Example 2.7 shows a Master cluster.

Example 2.7

The following Master cluster sends test lines to two different Slaves: **Dave** and **Anna**.

	1	2	3	4
1		**cluster**	Master example	
2		**author**	Housekeeper	
3				
4		**connect slave**	Dave	Anna
5	**Dave**		*number*	
6	**Dave**	**wash dishes**	26	
7	**Anna**		*room*	
8	**Anna**	**vacuum**	living room	
9	**Dave**		*room*	
10	**Dave**	**vacuum**	bedroom	
11		**disconnect slave**	Dave	
12			*room*	
13		**inspect**	living room	
14		**inspect**	bed room	
15	**Anna**	**vacuum**	staircase	
16	**Anna**	**take out garbage**		
17		**disconnect slave**	Anna	
18			*room*	
19		**inspect**	staircase	

3 REPORTS

3.1 Introduction

The built-in report generator generates a report that gives an overview of the results of the total test run. The internal format of this document is Rich Text Format (RTF).

3.2 Log

When executing the test clusters all relevant actions of the engine are logged. For every test run a log file is created; this log contains the actions of the main cluster, but also of its sub clusters and templates. This log file is used to generate the final report. Note that master and slave engines have separate log files; the engine's built-in report generator creates individual reports for masters and slaves.

The contents of a log file consists of lines. In each line the following items can be found: date of action, time of action, cluster line number, keyword specifying the type of action, and the action-specific parameters of this keyword. The name of a log file depends on the name of the processed cluster and time of processing; e.g., when the cluster `c:\testframe\cluster\cluster1.txt` is called on December 20, 2005 at 18:20:15 hours, a log file with the following name will be created: `cluster1@20051220_182015.log`. The name is based on the cluster name and the current date and time. If a cluster is called more than once in one second, a sequence number will added to the log file name.

3.3 Generating reports

3.3.1 Generating a report at the end of the test run

By calling the interface **tfe_GenerateReport** the Engine generates a report of the current test. This report is a Rich Text Format (RTF) file, containing different colors, fonts and pictures.

Generating a report can only be done after test processing is stopped—either after **tfe_StopEngine** is called or after the Stop button on the Engine Message Centre is pushed.

All the results of the test will be placed in one test report. The name and directory of this report are the ones specified in the Engine settings: in the section **[REPORT]** the keys **FileName** and **Path** respectively. This name and directory can be replaced—if so desired—when the test is started; see the description of **tfe_StartEngine**.

Only if during the test sub clusters were called with separate reports specified, will more than one report be generated—with the same **tfe_GenerateReport** call. The names of these reports are those specified in the clusters.

A specified report file need not exist beforehand—if it doesn't, it will be created. Should it, however, exist and be inaccessible—e.g. locked by a word processor—then the results will be written in a substitute report instead, which is placed in the working directory.

3.3.2 Generating a report outside a test run

As the previous paragraph stated: at the end of each test run a report can be generated for the current test; using the standard wrapper, this is always done. It is, however, also possible to generate a report for a test already finished—i.e. to generate a report for it outside of the test run. To do this, the interface function **tfe_GenerateReportFromLog** must be used.

In some ways, calling this function is much like calling **tfe_StartEngine**; only instead of the cluster file, the log file of a test must be specified. Out of this log file, a report will be generated by the Engine, giving it a specified name and using specified settings.

So, it is possible to generate several reports from the same test results, each report e.g. in a different language.

Also, note that a substitute report will be generated if the specified report file is inaccessible—the same as for reports generated using **tfe_GenerateReport**.

3.3.3 Adding logos to the RTF Reports

It is possible to place a user defined logo into the report. Place such a logo as RTF into the template directory, and call this file `logo.tpl`.

3.4 Report contents

The results of the test specific actions are placed in the report body; before it in the header, and after it in the footer, however, additional information is given.

3.4.1 Header

The report starts with a header. It contains a possible logo and the header information. Some header information is always written: license information and version and build of the engine. Other information comes from header words in the run's main cluster.

3.4.2 Footer

The footer of the report contains a summary of the test results; it contains several sections:

- Number of test lines.
  This number is the number of lines in the cluster that contain an action word.
- Succeeded test lines.
  This is the percentage of the test lines (as defined above) in which no error was reported and no check was failed.
- Error section.
  In this section the number of errors reported during a test run is given; if there are any , they are also given by type. At the end of the section the cluster line numbers, in which the errors occurred, are given.
- Check section.
  In this section the number of checks performed during a test run is given; furthermore, the numbers are given for those that were successful and those that failed (plus percentages). At the end of the section the cluster line numbers, in which the checks occurred, are given.
- Time section.

In this section the start time, end time, and total time used is printed.

4 THE ENGINE MESSAGE CENTRE

The Engine Message Centre (EMC) is the Engine's Graphical User Interface. With it, a test's progress and runtime statistics can be monitored. Furthermore, it can be used to influence a test: to pause, resume and stop it. This chapter discusses the features of the Engine Message Centre.

4.1 Starting the Engine Message Centre

There can be made a distinction between a *local* EMC and a *remote* EMC. A local EMC is one which runs on the *same* machine as the Engine it monitors. A remote EMC is one which runs on a machine *different* than the one on which its Engine runs—though connected to it via a network.

A local EMC can be started by the Engine itself. By calling the interface function **tfe_StartEngine** an EMC is started on the same machine, *if* the Engine is configured to do so. The setting to regulate this can be found in the Engine's ini file section **[GUI]**: the key **LaunchGUI**. If this key contains the value **Yes**, a local EMC is started; if **No**, it is not.

To start a remote EMC (or local EMC apart from the Engine), first of all, the executable *emc.exe, MessageCentre.dll* and *EngGUI.dll,* must be present on the machine. It must be called via the command prompt with the following command:

```
emc -h <hostname> -p <port number>
```

With this call, two command parameters must be given, preceded by their corresponding flags: the host name or IP address of the machine on which the Engine runs, and the TCP/IP port number used for this connection. Table 4.1 lists these.

parameter	*flag*	*parameter description*
<hostname>	-h	name or IP address by which the machine on which the Engine is running, is known in the network
<port number>	-p	the TCP/IP port number which the Engine has specified for this connection

Table 4.1 Remote EMC call command parameters

The port number is the one specified by the Engine to which the connection is made. It is set in its ini file, in the section **[GUI]** with the key **Port**. This port number must be unique, because it can not be used by any other process.

An Engine need not be connected to an EMC at all to process clusters. On the other hand, an Engine can also be connected to several EMC's. They are all started individually. On all these EMC's the same information is displayed.

In the Engine settings start-up conditions for all its EMC's can be determined. Whether or not its EMC's start up in a *detailed* state (with the key **Detail** in the section **[GUI]**,) and whether or not its EMC's are *on top* of other windows (with the key **OnTop** in the **[GUI]** section.) Furthermore, the settings determine the *language* of the EMC. This language is the same as the other language-dependent Engine features such as the report—either English, German, French or Dutch—and is set with the key **Language** in the section **[SYSTEM]**.

It is possible to synchronize the Engine and EMC. By giving the key **WaitForGUI** in the section **[SYSTEM]** the value **Yes**, the Engine is made to wait with processing until an EMC connects itself to it. To avoid deadlock, this waiting has a time-out period—five minutes. If the key **WaitForGUI** is set to **No**, the Engine continues processing whether or not EMC's are monitoring. In both cases, though, EMC's can be connected—or disconnected—at any time during the test.

Any EMC can be closed at any time during the test—by pushing the close button or killing the process some other way. However, when the interface function **tfe_StopEngine** is called, *all* EMC's connected to that Engine—local and remote—are sent the message to close themselves, which they will do.

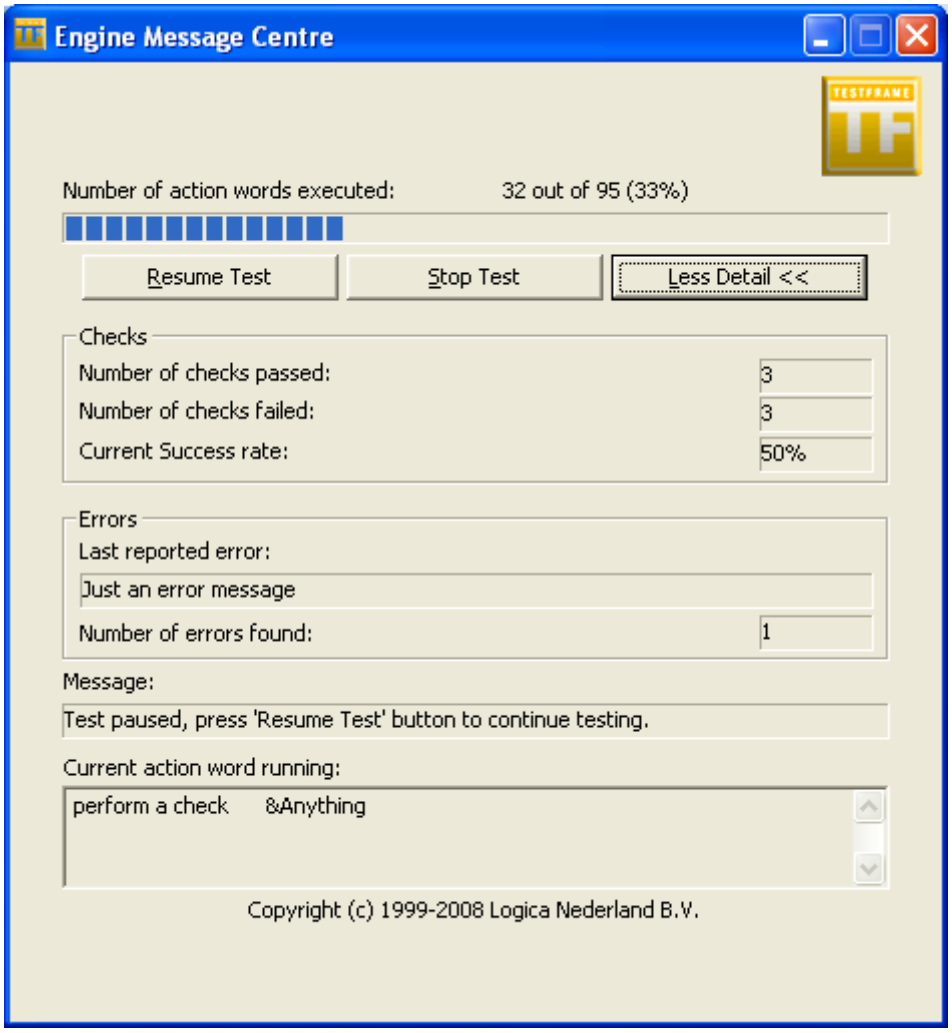

Figure 4.1 The Engine Message Centre in detailed state

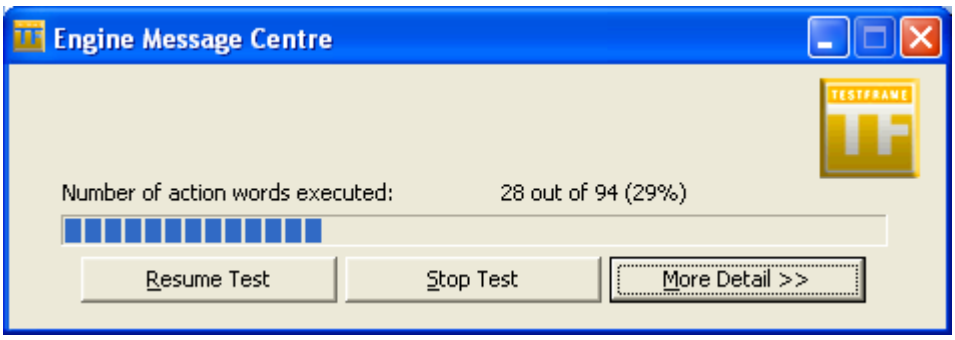

Figure 4.2 The Engine Message Centre in non-detailed state

4.2 Using the Engine Message Centre

To visualise what has been talked about, Figure 4.1 shows an EMC in detailed state, and Figure 4.2 shows one in non-detailed state. This paragraph will discuss what can be done with an EMC. Its features will be divided into *EMC window controls*, *Engine state controls*, and *run time information*. Figure 4.3 shows all EMC features, numbered with digits and letters.

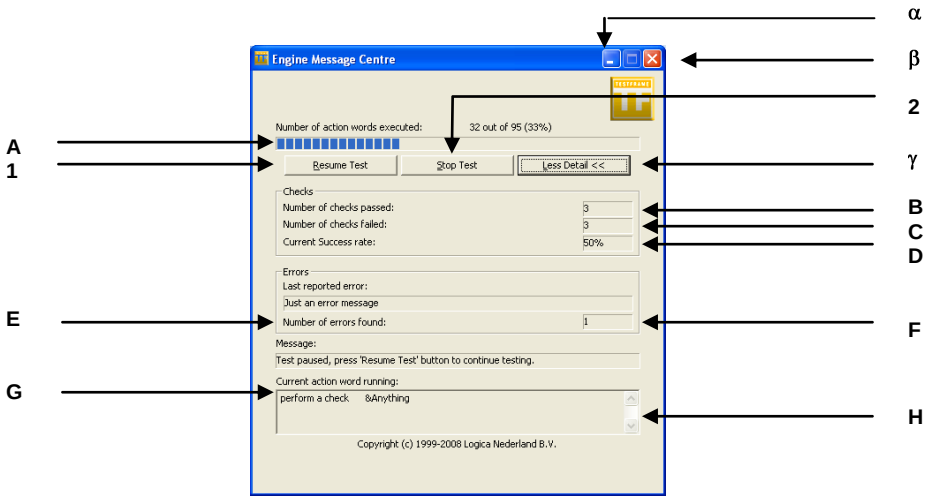

Figure 4.3 Features of the Engine Message Centre

4.2.1 Engine Message Centre window controls

There are three Engine Message Centre window controls. In Figure 4.3 they are referred to as α, β and γ.

α **Minimize button**
Pressing this button minimizes the EMC.

β **Close button**
Pressing this button closes the EMC.

γ **Detail button**
If the EMC is in a detailed state—like Figure 4.1—this button displays the text "**Less Detail <<**". In a non-detailed state—like Figure 4.2—the button reads "**More Detail >>**". Pressing this button will toggle the EMC: going from detailed to non-detailed or vice versa.

Engine state controls

The EMC has two buttons to influence the Engine's state. These are numbered **1** and **2** in Figure 4.3.

1 Pause/Resume button
When the Engine is processing a cluster, this button reads "**Pause Test**". Pressing it will indeed pause the Engine, making it wait for a resume message. If the Engine is paused, the text of the button is changed to "**Resume Test**"; this button is now not the **Pause**, but the **Resume** button. Only by pressing it the Engine is returned to a processing state.

If this button is pressed—either as Pause or as Resume—on any EMC, the text—and function—of the button changes on *all* EMCs connected to the Engine.

2 Stop button
When this button is pressed, the current test is stopped. It has the same effect as if the interface function **tfe_StopEngine** would be called. Furthermore, after pressing it, this and all other EMCs connected to the Engine are closed.

4.2.2 Run time information

Eight different types of run time information are displayed in the detailed state of the EMC. They are numbered **A** to **H** in Figure 4.3.

A Progress bar
This bar is a visual representation of the progress of the main cluster. Above it this progress is also printed, as the current cluster line number vs. the main cluster's total line number. This feature is the only run time information displayed when the EMC is in non-detailed state.

B Checks passed
The number of reported checks which have passed are printed here.

C Checks failed
This contains the number of reported checks which failed.

D Check success
The percentage of all checks which were passed up until now.

E Last error
The text of the last error message, regardless of error type.

F Number of errors
The total number of errors reported up until now.

G Message
A message sent by the Engine to the EMC to comment on the processing—e.g. the name of the current cluster file.

H Current test line

This box lists all valid test lines processed by the Engine; each new line is added to the bottom, and the focus is set to the current one. A scrollbar offers the possibility to go through the list.

5 SETTINGS AND INI FILE

The Engine has several user defined settings—the path of the log files for example, or of what characters the expression prefix consists, or whether the Engine runs in a Master mode or stand alone.

These settings must be configured in a ini file. This ini file can be given any name and placed in any directory—just as long as this file is corrected passed on to the Engine as the third parameter of the interface function to start the Engine processing a cluster: **tfe_StartEngine**.

If one or more of the keys contains a invalid value the Engine will not start—**tfe_StartEngine** will not process the cluster and will return 0. To the file error.log in the present working directory will be written an error message for each invalid key stating that its value is invalid and what values *are* acceptable. Be aware that each value is treated case sensitive.

The following paragraphs will describe the Engine's settings per section of the ini file. These settings are all the necessary entries in the Engine's ini file. Any user-added sections and/or keys are not guaranteed; users are recommended to place their own settings in another ini file.

5.1 The Cluster section

The Cluster section contains settings that are related to the cluster that is to be processed. In the ini file it is denoted by **[CLUSTER]**. The following two keys can be configured.

FileName
The value of this key is the name of the tab delimited cluster file that is to be processed. This name can be anything as long as it is not an empty string. When **tfe_StartEngine** is called with as its first parameter an empty string, this key's value is taken; otherwise it will be overwritten by the file name from the function call.

Path
This key contains the path of the tab delimited cluster file that is to be processed. As with the key **FileName**: it can be anything except empty, and its value is taken as the cluster path when **tfe_StartEngine** is called with an empty cluster parameter, otherwise overwritten.

5.2 The GUI section

The GUI section contains settings that initialises the Engine's own GUI: the Engine Message Centre. In the ini file it is denoted by **[GUI]**. Be aware that **System** section also contains a setting that is concerned with the GUI connection which, when used ill in combination with these settings, can lead to undesired results; more about that in paragraph 5.9. The following four keys can be configured in the GUI section.

Detail
The settings specifies how detailed the Engine Message Centre should be started. Its value can be either **Yes** or **No**. If it is **Yes**, the GUI contains full details; if **No**, only the progress bar is shown.

LaunchGUI

The value of this key must be either **Yes** or **No**. If it is **Yes**, the **tfe_StartEngine** call starts automatically an Engine Message Centre on the machine on which the Engine is running; if **No**, it does not.

Note that this settings is only concerned with *automatically* starting a *local* Engine Message Centre—if its value is **No**, it is still possible to start a remote GUI, or even a local one, but this then has to done by (manually) starting enginegui.exe.

OnTop
This key specifies whether or not the Engine Message Centre is to be the window on top. Its value must either be **Yes**—in which case it is on top—or **No**—in which case it isn't.

Port
The value of this key is TCP/IP port number for the GUI connection. It must be a number from 1024 to 65535, and be unique. So beware it's not the same as the Master/Slave port number.

5.3 The Keep section

The Keep section contains settings that are related to the keep file. In the ini file it is denoted by **[KEEP]**. The following two keys can be configured.

FileName
This key's value is the name of the keep file. It must not be an empty string.

Path
This setting specifies the path of the keep file. It, too, must not be empty.

5.4 The License section

The License section contains settings that are related to the license file. In the ini file it is denoted by **[LICENSE]**. The following four keys are present in this setting, but only Path is functional—the other three are commentary, showing the license information of the last test run with this ini file.

Company
This key specifies the company name from the license file. This setting is non-functional, commentary only. When starting the Engine its value is overwritten with the one taken from the license.key file.

Name
This key's value contains the license name. As with **Company**, also a commentary setting, and overwritten when the Engine's started.

Path
This settings contains the path of the license file—license.key. Make sure this path is correct, otherwise the Engine won't find the license file and consequently not start—an error message will in that case be written to the error log.

SerialNumber
The value of this key is the license serial number. Again, as **Company** and **Name**, a commentary setting that'll be overwritten.

5.5 The Log section

The Log section specifies the log files information. In the ini file it is denoted by **[LOG]**. It contains one setting.

Path
This setting specifies the path of the log files. It must not be an empty string.

5.6 The Master/Slave section

The Master/Slave section contains settings that are related to the Master/Slave concept. In the ini file it is denoted by **[MASTERSLAVE]**. The following two keys can be configured.

Mode
This key specifies the mode in which in the Engine should run: stand alone mode in which the Engine makes no use of the Master/Slave concept or runs only as Slave, or Master mode in which the Engine runs a Master cluster. In the former case this key's value must be **Standalone**, in the latter one **Master**.

Port
The value of this key is TCP/IP port number for the Master/Slave connection. It must be a number from 1024 to 65535, and be unique. So beware it's not the same as the GUI port number.

To this port number must be referred in the tmsp URL of slave clusters when connecting to this Engine as their Master.

5.7 The Report section

The Report section contains settings that are related to the report that is to be generated for the processed test. In the ini file it is denoted by **[REPORT]**. The following five keys can be configured.

Arguments
This key specifies whether or not all arguments of a test line are to be printed in the report. If its value is **Yes**, the complete test line will be printed; if it is **No**, only the name of the test line's action word will be printed in the report.

FileName
The value of this key is the name of the report file. This name can be anything as long as it is not an empty string. When **tfe_StartEngine** is called with as its second parameter an empty string, this key's value is taken; otherwise it will be overwritten by the file name from the function call.

Overwrite
This key's value must be either **Yes** or **No**. If it is **Yes**, the specified report file will be overwritten; if it is **No**, the previous report with the same file name will be saved as back-up before the report is written.

Path
This key contains the path of the report file. As with the key **FileName**: it can be anything except empty, and its value is taken as the report path when **tfe_StartEngine** is called with an empty report parameter, otherwise overwritten.

TemplatePath

This setting specifies the path of the templates that are used in the report. At the moment this is only the header logo—logo.tpl. The **TemplatePath**'s value must not be empty.

5.8 The Snap section

The Snap section contains settings that are related to the snap file. In the ini file it is denoted by **[SNAP]**. The following two keys can be configured.

FileName
This key's value is the name of the snap file. It must not be an empty string.

Path
This setting specifies the path of the snap file. It, too, must not be empty.

5.9 The System section

The System section contains settings that are related to the way the Engine processes data. In the ini file it is denoted by **[SYSTEM]**. It consists of fourteen keys; one of commentary nature—**EngineVersion**—and eight prefixes—**ContText** is counted for this purpose as a prefix—which must all be non-empty and unique.

AnythingPrefix
This key contains the characters that make up the anything prefix. Almost anything is allowed—as long as it is non-empty and unique.

ContText
This key value defines the continue token. It, too, must be non-empty and unique.

DatePrefix
This setting specifies the date prefix; again, non-empty and unique.

EmptyPrefix
This key's value holds the characters of the empty command prefix.

EngineBuild
This setting contains the build number of the Engine. It is a non-functional, commentary setting which is overwritten when the Engine is started.

EngineVersion
This setting contains the version of the Engine. Just as **EngineBuild**, it's a commentary setting.

ExpressionPrefix
This key defines the expression prefix.

KeepPrefix
This setting's value contains the name of the keep prefix.

Language
This setting specifies the language that should be used in the Engine Message Centre and the report texts. Four languages are supported: English, German, French and Dutch whose corresponding setting's values are **GB**, **DE**, **FR** and **NL** respectively.

NotEmptyPrefix
The value of this setting specifies the not-empty command prefix.

Precision
This key specifies how floating point numbers are to be converted to strings in expressions. The value consists of two parts:

- A number, the precision, that specifies the number of digits to be printed after the decimal point for e, E or f conversions, or the number of significant digits for g or G conversion. The precision may also be specified as *, in which case the value is computed by taking the precision of the float to be converted. The maximum allowed precision is 15; any precision found larger is set to 15.

- A conversion character; all of which are described in the table below.

character	converted to
f	decimal notation of the form [-]*mmm.ddd* where the number of *d*'s is specified by the precision. A precision of 0 suppresses the decimal point. If the number of *m*'s exceeds 15, informations may be lost in the conversion.
e, E	decimal notation of the form [-]*m.ddddd* e±*xxx* or [-]*m.ddddd* E±x*xx* , where the number of d's is specified by the precision. A precision of 0 suppresses the decimal point.
g, G	e or E is used if the exponent is less than –4 or greater than or equal to the precision; otherwise f is used. Trailing zero's and a trailing decimal point are not printed. a significance of 0 is treated like 1: one significant digit.

RedundantEscapeChars
This setting's value holds the redundant escape characters. A redundant escape character is a character that is ignored by the Engine if there is a single occurrence of it, but treated as one such character if there two of them in a row. This key's value can be empty (no redundant escape characters,) or filled with any character a user wishes to let the Engine treat as such.

If one uses tab separated text files that are generated by Microsoft® Excel, the double quotation mark—the character "—can be added to this setting, since these characters are escaped by Excel and added around cells in which they or commas occur.

SnapPrefix
The value of this key is the sequence of characters that make up the snap prefix.

SpacesPrefix
This setting's value defines the spaces command prefix.

WaitForGUI
This key specifies whether or not the Engine should wait with processing until an Engine Message Centre—local or remote—attaches itself to the Engine. The allowed values are **Yes** and **No** for waiting and not waiting respectively.

The Engine waits—if the value is **Yes**—for a maximum of five minutes before it starts processing, but in the meantime the system is blocked. Be aware that when this setting's value is **Yes** and the value of the key **LaunchGUI** in the GUI section is **No**, the Engine expects—for five minutes at least—that a remote Engine Message Centre will attach itself before it may start processing.

5.10 The Template section

The Template section specifies the template files information. In the ini file it is denoted by **[TEMPLATE]**. It contains one setting.

Path
This setting specifies the path of the template files. It must not be an empty string.

5.11 An example of an Engine ini file

Below is given an example of an ini file that can be used for the processing of a stand alone cluster.

Example 5.1

```
[CLUSTER]
FileName=Example.txt
Path=C:\TestFrame\Cluster\

[GUI]
Detail=Yes
LaunchGUI=Yes
OnTop=No
Port=4000

[KEEP]
FileName=keep.ini
Path=c:\TestFrame\Keep\

[LICENSE]
Company=Logica
Name=TestFrame Engine Team
Path=c:\TestFrame\
SerialNumber=550-000001-0-01

[LOG]
Path=c:\TestFrame\Log\

[MASTERSLAVE]
Mode=Standalone
Port=4100
```

```
[REPORT]
Arguments=Yes
FileName=ExampleReport.rtf
Overwrite=Yes
Path=C:\TestFrame\Report\
TemplatePath=c:\TestFrame\Report\Template\

[SNAP]
FileName=snap.ini
Path=c:\TestFrame\Snap\

[SYSTEM]
AnythingPrefix=&Anything
ContText=&Cont
DatePrefix=&Date()
EmptyPrefix=&Empty
EngineBuild=1
EngineVersion=2008.04
ExpressionPrefix=&
KeepPrefix=&Keep()
Language=GB
NotEmptyPrefix=&NotEmpty
Precision=6g
RedundantEscapeChars=
SnapPrefix=&Snap()
SpacesPrefix=&Spaces()
WaitForGUI=Yes

[TEMPLATE]
Path=c:\TestFrame\Template\
```

6 BUILT-IN ACTION WORDS

Built-in action words are action words meant for the Engine to perform a certain action. They do not correspond with a user defined action word function. The TestFrame Engine has the following built-in action words:

report header

cluster
author
date
version
document
sheet

structuring

scenario
section
test condition
test case

subcluster

do cluster

template

define template
end template
declare template
template prototype

cluster variables

export variable
import variable

master/slave

connect slave
disconnect slave

flow of control

If
Else
else if
end if
Repeat
end repeat
While
end while

Utility

Set

6.1 author

Prototype

	1	2
1	**author**	Author name

Parameter

#	*prototype name*	*Description*
1	Author name	name of the cluster's author

Built-in synonyms **auteur**
Autor

Action word type header word

Description
This header word logs its parameter as the cluster's author name. When a report is generated, it is placed in the report header after the 'Cluster author' tag.

6.2 cluster

Prototype

	1	2
1	**cluster**	Cluster name

Parameter

#	*prototype name*	*Description*
1	Cluster name	name of the cluster

Built-in synonym **Cluster**

Action word type header word

Description
This header word logs its parameter as the cluster's name. When a report is generated, it is placed in the report header after the 'Cluster name' tag.

6.3 connect slave

Prototype

	1	2	3	...	n + 2
1		**connect slave**	First slave	...	n–th slave

Parameters

#	*prototype name*	*Description*
1	First slave	name of slave to connect
n	n–th slave	name of another slave to connect

Built-in synonyms **connecter esclave**
verbind slaaf
Verbinde Slave

Action word type Master/Slave synchronisation

Description
This action word establishes the connection between a Master and one or several of its Slaves. It must be used in a Master cluster, and, consequently, is only useful in Master mode—hence the action word's place in the second argument in its prototype above. For this action to succeed, the Slaves whose names are placed as the action word's parameters must have connected themselves to the Master before this action is called, or must do so within a time-out period of five minutes.

6.4 date

Prototype

	1	2
1	**date**	Cluster date

Parameter

#	*prototype name*	*Description*
1	Cluster date	date of the cluster

Built-in synonyms **Datum**
datum

Action word type header word

Description
This header word logs its parameter as the cluster's date. When a report is generated, it is placed in the report header after the 'Cluster date' tag.

6.5 declare template

Prototype

	1	2	3
1	**declare template**	Template name	Template file

Parameters

#	*prototype name*	*Description*
1	Template name	name of the template to declare
2	Template file	file in which the cluster lines of the template are placed

Built-in synonyms **déclaration de template**
declareer template
Deklariere Template

Action word type template action word

Description
This action word declares that a previously defined template will be used in a cluster, and in which file its lines can be found. For this action to succeed, the declared name of the template must correspond with the one in the specified file's template prototype. If no path is specified for the template file, the template directory specified in the settings is taken.

6.6 define template

Prototype

	1	2	3	...	n + 2
1	**define template**	Template name	First temp. param.	...	n-th temp. param.

Parameters

#	*prototype name*	*Description*
1	Template name	name of the template to define
2	First temp. param.	name of first template parameter
n	n–th temp. param.	name of n–th template parameter

Built-in synonyms **definieer template**
definiëer template
Definiere Template
définition de template

Action word type template action word

Description
This action word starts the definition of a template and declares its prototype. This prototype must contain a unique template name (this action word's first parameter) by which it can be called in a cluster; furthermore, it can contain no, one or more template parameters, which can be used as cluster variables within the template, and which are assigned during the template call.

6.7 disconnect slave

Prototype

	1	2	3	...	n + 2
1		**disconnect slave**	First slave	...	n-th slave

Parameters

#	*prototype name*	*Description*
1	First slave	name of slave to disconnect
n	n-th slave	name of another slave to disconnect

Built-in synonyms **Beende Slave**
déconnecter esclave
verbreek slaaf

Action word type Master/Slave synchronisation

Description
This action word ends the connection between a Master and one or several of its Slaves. It must be used in a Master cluster, and, consequently, is only useful in Master mode—hence the action word's place in the second argument in its prototype above. In order to disconnect a specified Slave, it must first have been connected.

6.8 do cluster

Prototype

	1	2	3
1	**do cluster**	Cluster file	Report file

Parameters

#	*prototype name*	*Description*
1	Cluster file	(sub)cluster file to process
2	Report file	file in which to write the (sub)cluster's results

Built-in synonyms **exécuter cluster**
Führe Cluster aus
voer cluster uit

Action word type sub cluster action word

Description
This action word pauses the processing of the current cluster and starts processing a sub cluster. This sub cluster is specified by the action word's first parameter. If no path is specified for the sub cluster, the cluster directory from the settings is taken; if a TestFrame Master/Slave Protocol (tmsp) URL is used, the sub cluster is not taken from a file, but a connection as Slave Engine to a Master is initiated. The second parameter specifies the report file in which the sub cluster's results are to be written; if no directory is specified, the setting's report path is taken; if the entire parameter is empty, its results are placed in the calling cluster's report.

6.9 document

Prototype

	1	2
1	**document**	Document name

Parameter

#	*prototype name*	*Description*
1	Document name	name of the cluster's document

Built-in synonym **Document**

Action word type header word

Description
This header word logs its parameter as the cluster's document name. When a report is generated, it is placed in the report header after the 'Document' tag.

6.10 else

Prototype

	1	2
1	**else**	

Parameters
This action word has no parameters.

Action word type flow of control

Description
This action starts the alternative of a selection statement. It must be preceded by an **if** action word, and followed by an **end if** action word. The test lines between this test line and that of the corresponding end if are processed if the last condition of its selection statement is evaluated as false; otherwise, these lines are skipped.

6.11 else if

Prototype

	1	2	...	n + 1
1	**else if**	Condition argument 1	...	Condition argument n

Parameters

#	*prototype name*	*description*
1	Condition argument 1	first argument of the condition
n	Condition argument n	last argument of the condition

Action word type flow of control

Description
This action word starts an alternative condition within a multi-way decision statement. It must be preceded by an **if** action word, and followed by an **end if** action word. The arguments following the action word state the condition; this condition is evaluated if all preceding conditions in its selection statement are *false*; if this condition is evaluated as *true*, the cluster lines directly following this test line are processed, otherwise skipped.

6.12 end if

Prototype

	1	2
1	**end if**	

Parameters
This action word has no parameters.

Action word type flow of control

Description
This action word ends a selection statement. It must be preceded by an **if** action word.

6.13 end repeat

Prototype

	1	2
1	**end repeat**	

Parameters
This action word has no parameters.

Action word type flow of control

Description
This action word marks the end of an unconditional loop. It must be preceded by a **repeat** action word.

6.14 end template

Prototype

	1	2
1	**end template**	

Parameters
This action word has no parameters.

Built-in synonyms **einde template**
Ende Template
fin de template

Action word type template action word

Description
This action word ends the definition of a template. It must be preceded by a **define template** action word, and all cluster lines in between the two are taken as the template's lines.

6.15 end while

Prototype

	1	2
1	**end while**	

Parameters
This action word has no parameters.

Action word type flow of control

Description
This action word marks the end of a conditional loop. It must be preceded by a **while** action word.

6.16 export variable

Prototype

	1	2	...	n + 1
1	**export variable**	Cluster variable 1	...	Cluster variable n

Parameters

#	*prototype name*	*description*
1	Cluster variable 1	name of a cluster variable to export
n	Cluster variable n	name of another cluster variable to export

Built-in synonyms **exporteer variabele**
exporter variable
Exportiere Variable

Action word type action word for data transfer

Description
This action word exports data to the keep file. The one or more cluster variables, whose names are specified as the action word's parameters, are written to the [EXPORT] section of the keep file as keys; as values are written the variable's values at the moment of export.

6.17 if

Prototype

	1	2	...	n + 1
1	**if**	Condition argument 1	...	Condition argument n

Parameters

#	*prototype name*	*description*
1	Condition argument 1	first argument of the condition
n	Condition argument n	last argument of the condition

Action word type flow of control

Description
This action word starts a selection statement. The arguments following the action word state the condition; if this condition is evaluated as *true*, the cluster lines directly following this test line are processed, otherwise skipped.

6.18 import variable

Prototype

	1	2	...	n + 1
1	**import variable**	Cluster variable 1	...	Cluster variable n

Parameters

#	*prototype name*	*description*
1	Cluster variable 1	name of a cluster variable to import
n	Cluster variable n	name of another cluster variable to import

Built-in synonyms **importeer variabele**
importer variable
Importiere Variable

Action word type action word for data transfer

Description
This action word imports data from the keep file. The one or more cluster variables whose names are specified as the action word's parameters, are read from the [EXPORT] section of the keep file; the names are taken as the keys, and the values from the keep file are made the cluster variable's values. If the cluster variable has not been used before in the cluster, it is created; if it is not present in the keep file, the operation will fail.

6.19 repeat

Prototype

	1	2
1	**repeat**	Number of iterations

Parameters

#	*prototype name*	*description*
1	Number of iterations	number of times the cluster lines in the unconditional loop are to be processed

Action word type flow of control

Description
This action word starts an unconditional iteration statement. Its parameter states the number the times the cluster lines in this loop are to be processed; it must be a non-negative integer (or an expression which evaluates to one.) This line must be followed by an **end repeat** action word, which states the loop's end.

6.20 scenario

Prototype

	1	2
1	**scenario**	Scenario description

Parameter

#	*prototype name*	*description*
1	Scenario description	description of the new scenario

Built-in synonyms **scénario**
Testszenario

Action word type cluster structuring

Description
This action word marks the start of a new scenario, one of the structure elements. Its parameter states the description—name, number, etc. If a scenario was already started in the same cluster, this action word also marks the end of this previous one.

6.21 section

Prototype

	1	2
1	**section**	Section description

Parameter

#	*prototype name*	*description*
1	Section description	description of the new section

Built-in synonyms **sectie**
Testabschnitt

Action word type cluster structuring

Description
This action word marks the start of a new section, one of the structure elements. Its parameter states the description—name, number, etc. If a section was already started in the same cluster, this action word also marks the end of this previous one.

6.22 set

Prototype

	1	2	..	n + 1
1	**set**	Parameter 1	..	Parameter n

Parameter

#	*prototype name*	*description*
1	Parameter 1	first argument to process
n	Parameter n	n-th argument to process

Action word type utility

Description
This action word has no functionality other than to offer a way to let the Engine process test line arguments without these being used directly for specific actions. It can be used to process argument commands; expressions and assignments of cluster variables in particular.

6.23 sheet

Prototype

	1	2
1	**sheet**	Sheet name

Parameter

#	*prototype name*	*description*
1	Sheet name	name of the cluster's sheet

Built-in synonyms **Arbeitsblatt**
blad
feuille de calcul

Action word type header word

Description
This header word logs its parameter as the cluster's sheet name. When a report is generated, it is placed in the report header after the 'Sheet' tag.

6.24 template prototype

Prototype

	1	2	3	...	n + 2
1	**template prototype**	Template name	First temp. param.	...	n-th temp. param.

Parameters

#	*prototype name*	*description*
1	Template name	name of the template
2	First temp. param.	name of first template parameter
n	n–th temp. param.	name of n–th template parameter

Built-in synonyms
This action word has no built-in synonyms.

Action word type template action word

Description
This action word states the prototype of a template. After a template is successfully defined, the Engine writes this action word at the beginning of a template file; it is not meant to be placed in clusters by users.

6.25 test case

Prototype

	1	2
1	**test case**	Test case description

Parameter

#	*prototype name*	*description*
1	Test case description	description of the new test case

Built-in synonyms **cas de test**
testcase
Testfall
testgeval

Action word type cluster structuring

Description
This action word marks the start of a new test case, one of the structure elements. Its parameter states the description—name, number, etc. If a test case was already started in the same cluster, this action word also marks the end of this previous one.

6.26 test condition

Prototype

	1	2
1	**test condition**	Test condition description

Parameter

#	*prototype name*	*description*
1	Test condition description	description of the new test condition

Built-in synonyms **condition de test**
Testbedingung
testconditie
testcondition

Action word type cluster structuring

Description
This action word marks the start of a new test condition, one of the structure elements. Its parameter states the description—name, number, etc. If a test condition was already started in the same cluster, this action word also marks the end of this previous one.

6.27 version

Prototype

	1	2
1	**version**	Cluster version

Parameter

#	*prototype name*	*description*
1	Cluster version	version of the cluster

Built-in synonyms **versie**
Version

Action word type header word

Description
This header word logs its parameter as the cluster's version. When a report is generated, it is placed in the report header after the 'Cluster version' tag.

6.28 while

Prototype

	1	2	...	n + 1
1	**while**	Condition argument 1	...	Condition argument n

Parameters

#	*prototype name*	*description*
1	Condition argument 1	first argument of the condition
n	Condition argument n	last argument of the condition

Action word type flow of control

Description

This action word starts a conditional iteration statement. This line must be followed by an **end while** action word, which states the loop's end; all cluster lines in between are part of the loop. The arguments following the **while** action word state the condition; if this condition is evaluated as *true*, the loop is processed, and the condition is evaluated again; once the condition becomes *false* the loop is skipped.

7 INTERFACE FUNCTIONS OF THE ENGINE

The Engine exports its interface functions as C functions; their prototypes listed below are therefore given in C syntax. The data types used are integers (**`int`**) and zero-terminated character strings (**`char*`**); if one wants to use these functions, one should be able to convert these types to ones available in the test software's programming language—e.g. C++, Java or any of the test tool languages: WinRunner, QARun, Quick test Pro, Rational Robot.

Below, all of the interfaces are listed.

action word registration
tfe_RegisterActionWord
tfe_RegisterHeaderWord
tfe_CreateActionWordSynonym
tfe_UnregisterActionWord
tfe_UnregisterHeaderWord
tfe_UnregisterAllActionWords

state control
tfe_StartEngine
tfe_StopEngine
tfe_ResetEngine

report generating
tfe_GenerateReport
tfe_GenerateReportFromLog

test flow
tfe_ProcessNextLine

run time information
tfe_GetActionWord
tfe_GetActionWordFunction
tfe_GetArgument
tfe_GetParameter
tfe_GetNumberOfArguments
tfe_GetNumberOfParameters
tfe_GetArgumentDescription
tfe_GetParameterDescription
tfe_GetTestCase
tfe_GetTestCondition
tfe_GetSection
tfe_GetScenario
tfe_GetClusterFile
tfe_GetLineNumber
tfe_GetNumberOfErrors
tfe_GetLatestError
tfe_GetCopyright
tfe_GetVersion
tfe_GetBuild
tfe_GetLogFile

reporting
tfe_ReportComment
tfe_ReportError
tfe_ReportCheck

checking
tfe_CheckString
tfe_CheckArgument
tfe_CheckParameter

cluster variables
tfe_KeepArgument
tfe_KeepParameter

GUI
tfe_DisplayMessage

construction/destruction (UNIX only)
tfe_CreateEngine
tfe_DeleteEngine

7.1 tfe_CheckArgument

Prototype

```
int tfe_CheckArgument( int nArgument, char* pszRecorded )
```

Parameters

#	type	prototype name	description
1	int	nArgument	index number of argument to perform check on
2	char*	pszRecorded	string to compare with specified argument's value

Return values

type	possible values	description
int	1	check passed
	0	check failed

Description

This function checks the value of a specified argument against a specified string, and logs the result. A check is considered passed if all characters of both strings match; only if the argument contains one of the check commands (&Empty, &NotEmpty or &Anything) another type of comparison is made.

7.2 tfe_CheckParameter

Prototype

```
int tfe_CheckParameter( int nParameter, char* pszRecorded )
```

Parameters

#	type	prototype name	description
1	int	nParameter	index number of parameter to perform check on
2	char*	pszRecorded	string to compare with specified parameter's value

Return values

type	possible values	description
int	1	check passed
	0	check failed

Description

This function checks the value of a specified action word parameter against a specified string. Its functionality is the same as that of tfe_CheckArgument, the only difference being that the first function parameter specifies the index of an action word parameter instead of an argument.

7.3 tfe_CheckString

Prototype

```
int tfe_CheckString( char* pszDescription, char* pszExpected,
                     char* pszRecorded )
```

Parameters

#	type	prototype name	description
1	char*	pszDescription	description of the check
2	char*	pszExpected	string with expected value
3	char*	pszRecorded	string with recorded value

Return values

type	possible values	description
int	1	check passed
	0	check failed

Description

This function checks the values of two specified strings against each other, and logs the result. A check is considered passed if all characters of both strings match. The specified description is written in the report as header above the result of the check.

7.4 tfe_CreateActionWordSynonym

Prototype

```
int tfe_CreateActionWordSynonym( char* pszActionWord,
                                 char* pszSynonym )
```

Parameters

#	type	prototype name	description
1	char*	pszActionWord	action word for which to create synonym
2	char*	pszSynonym	synonym to register

Return values

type	possible values	description
int	1	synonym created and registered
	0	synonym could not be created and registered

Description

This function creates a synonym for an action word; this can be a user defined or a built-in action word. In order to succeed the synonym must be unique and the action word must already be known.

7.5 tfe_CreateEngine

Prototype
```
int tfe_CreateEngine( )
```

Parameters
This function has no parameters.

Return values

type	*possible values*	*description*
`int`	1	creation successful
	0	creation failed

Description
This function creates an instance of the Engine. N.B. Unix only.

7.6 tfe_DeleteEngine

Prototype
```
int tfe_DeleteEngine( )
```

Parameters
This function has no parameters.

Return values

type	*possible values*	*description*
`int`	1	deletion successful
	0	deletion failed

Description
This function destructs the Engine instance. N.B. Unix only.

7.7 tfe_DisplayMessage

Prototype
```
int tfe_DisplayMessage( char* pszMessage )
```

Parameter

#	*type*	*prototype name*	*description*
1	`char*`	pszMessage	message to be displayed on a screen

Return values

type	*possible values*	*description*
`int`	1	message displayed
	0	message could not be displayed

Description
This function displays a specified message string on a window. This window is created and displayed on every machine, which has a GUI link to the Engine (Engine Message Centre.) The Engine is paused, until the OK button on the window is pressed on the window is closed.

7.8 tfe_GenerateReport

Prototype
```
int tfe_GenerateReport( )
```

Parameters
This function has no parameters.

Return values

type	*possible values*	*description*
`int`	1	report(s) created
	0	report(s) could not be created

Description
This function creates one or more reports for the current test; the main report, specified in the settings, and the possible reports of sub cluster results, if and as specified in the cluster files. If a report can not be created, the results will be written to a substitute report placed in the working directory.

7.9 tfe_GenerateReportFromLog

Prototype
```
int tfe_GenerateReportFromLog( char* pszLog, char* pszReport,
                               char* pszIniFile )
```

Parameters

#	*type*	*prototype name*	*description*
1	`char*`	pszLog	log file to generate report from
2	`char*`	pszReport	name of report in which to write the results
3	`char*`	pszIniFile	ini file containing the Engine settings

Return values

type	*possible values*	*description*
`int`	1	report(s) created
	0	report(s) could not be created

Description
This function creates one or more reports from a specified log file; one for the specified log, and one for each log file linked to; these log files must (still) exist. The report is configured using the settings from the specified ini file. If a report can not be created, the results will be written to a substitute report placed in the working directory.

7.10 tfe_GetActionWord

Prototype
```
char* tfe_GetActionWord( )
```

Parameters
This function has no parameters.

Return values

type	*description*
`char*`	name of action word

Description
This function returns the name of the current test line's action word. If there is no current test line—cluster processing is not started or has finished—an empty string is returned.

7.11 tfe_GetActionWordFunction

Prototype
```
char* tfe_GetActionWordFunction( )
```

Parameters
This function has no parameters.

Return values

type	*description*
`char*`	name of action word function

Description
This function returns the name the action word function registered with the current test line's action word—i.e. the string entered in **tfe_RegisterActionWord** as second parameter. If there is no current test line—cluster processing is not started or has finished—an empty string is returned.

7.12 tfe_GetArgument

Prototype
```
char* tfe_GetArgument( int nArgument )
```

Parameter

#	*type*	*prototype name*	*description*
1	`int`	nArgument	index number of argument

Return values

type	*description*
`char*`	value of specified argument

Description
This function returns the value of a specified argument of the current test line. If the specified number is out of range, or if there is no current test line—cluster processing is not started or has finished—an empty string is returned.

7.13 tfe_GetArgumentDescription

Prototype
```
char* tfe_GetArgumentDescription( int nArgument )
```

Parameter

#	*type*	*prototype name*	*description*
1	`int`	nArgument	index number of argument

Return values

type	*description*
`char*`	description of specified argument

Description
This function returns the description of a specified argument of the current test line. If the specified number is out of range, or if there is no current test line—cluster processing is not started or has finished—an empty string is returned.

7.14 tfe_GetBuild

Prototype
```
char* tfe_GetBuild( )
```

Parameters
This function has no parameters.

Return values

type	*description*
`char*`	build number of the Engine

Description
This function returns a string containing the build number of the Engine.

7.15 tfe_GetClusterFile

Prototype
```
char* tfe_GetClusterFile( )
```

Parameters
This function has no parameters.

Return values

type	*description*
`char*`	name (including path) of the current cluster file

Description
This function returns the name of the cluster file, which the Engine is currently processing. If no cluster is being processed, an empty string is returned.

7.16 tfe_GetCopyright

Prototype
```
char* tfe_GetCopyright( )
```

Parameters
This function has no parameters.

Return values

type	*description*
`char*`	copyright information of the Engine

Description
This function returns a string containing the copyright information of the Engine.

7.17 tfe_GetLatestError

Prototype
```
char* tfe_GetLatestError( )
```

Parameters
This function has no parameters.

Return values

type	*description*
`char*`	text of the latest error

Description
This function returns the text of the latest error—either reported by the user (via **tfe_ReportError**) by the Engine itself (a cluster error.) If no error has been found during the test run, an empty string is returned.

7.18 tfe_GetLineNumber

Prototype
```
int tfe_GetLineNumber( )
```

Parameters
This function has no parameters.

Return values

type	*possible values*	*description*
`int`	> 0	line number
	0	no current test line

Description
This function returns the number of the current test line. If there is no current test line—cluster processing is not started or has finished—zero is returned.

7.19 tfe_GetLogFile

Prototype
```
char* tfe_GetLogFile( )
```

Parameters
This function has no parameters.

Return values

type	*description*
`char*`	name (including path) of the current log file

Description
This function returns the name of the cluster file, which the Engine is currently writing to.

7.20 tfe_GetNumberOfArguments

Prototype
```
int tfe_GetNumberOfArguments( )
```

Parameters
This function has no parameters.

Return values

type	*possible values*	*description*
`int`	> 0	number of arguments
	0	no current test line

Description
This function returns the number of arguments of which the current test line consists; this also includes the argument with the action word name, and—when in Master mode—the argument of the Slave column. If there is no current test line—cluster processing is not started or has finished—zero is returned.

7.21 tfe_GetNumberOfErrors

Prototype
```
int tfe_GetNumberOfErrors( )
```

Parameters
This function has no parameters.

Return values

type	*possible values*	*description*
`int`	>= 0	number of errors

Description
This function returns the number of errors found during the test run. If errors have been found, or if no cluster is being processed, zero is returned.

7.22 tfe_GetNumberOfParameters

Prototype
```
int tfe_GetNumberOfParameters( )
```

Parameters
This function has no parameters.

Return values

type	*possible values*	*description*
`int`	> 0	number of parameters
	0	no current test line

Description
This function returns the number of parameters specified for the current action word. Note that this differs from the number of arguments by the excluding the action word, and—when in Master mode—the argument of the Slave column. If there is no current test line—cluster processing is not started or has finished—zero is returned.

7.23 tfe_GetParameter

Prototype
```
char* tfe_GetParameter( int nParameter )
```

Parameter

#	type	prototype name	description
1	int	nParameter	index number of parameter

Return values

type	description
char*	value of specified parameter

Description
This function returns the value of a specified parameter of the current action word. If the specified number is out of range, or if the action word has no parameters, or if there is no current test line—cluster processing is not started or has finished—an empty string is returned. Note that its functionality is the same as that of tfe_GetArgument; the difference is that the first parameter states the index of an action word parameter, not a test line argument.

7.24 tfe_GetParameterDescription

Prototype

```
char* tfe_GetParameterDescription( int nParameter )
```

Parameter

#	type	prototype name	description
1	int	nArgument	index number of parameter

Return values

type	description
char*	description of specified parameter

Description
This function returns the description of a specified parameter of the current action word. If the specified number is out of range, or if the action word has no arguments, or if there is no current test line—cluster processing is not started or has finished—an empty string is returned.

7.25 tfe_GetScenario

Prototype

```
char* tfe_GetScenario( )
```

Parameters
This function has no parameters.

Return values

type	description
char*	name of scenario

Description
This function returns the name of the latest scenario in the current cluster file. If no scenario has been started, or no cluster is being processed, an empty string is returned.

7.26 tfe_GetSection

Prototype
```
char* tfe_GetSection( )
```

Parameters
This function has no parameters.

Return values

type	*description*
`char*`	name of section

Description
This function returns the name of the latest section in the current cluster file. If no section has been started, or no cluster is being processed, an empty string is returned.

7.27 tfe_GetTestCase

Prototype
```
char* tfe_GetTestCase( )
```

Parameters
This function has no parameters.

Return values

type	*description*
`char*`	name of test case

Description
This function returns the name of the latest test case in the current cluster file. If no test case has been started, or no cluster is being processed, an empty string is returned.

7.28 tfe_GetTestCondition

Prototype
```
char* tfe_GetTestCondition( )
```

Parameters
This function has no parameters.

Return values

type	*description*
`char*`	name of test condition

Description
This function returns the name of the latest test condition in the current cluster file. If no test condition has been started, or no cluster is being processed, an empty string is returned.

7.29 tfe_GetVersion

Prototype
```
char* tfe_GetVersion( )
```

Parameters
This function has no parameters.

Return values

type	*description*
`char*`	version of the Engine

Description
This function returns a string containing the version of the Engine.

7.30 tfe_KeepArgument

Prototype
```
int tfe_KeepArgument( int nArgument, char* pszValue )
```

Parameters

#	*type*	*prototype name*	*description*
1	`int`	nArgument	index number of argument with keep command
2	`char*`	pszValue	value to keep

Return values

type	*possible values*	*description*
`int`	1	value successfully kept
	0	value could not be kept

Description
This function sets the value of a cluster variable in a specified argument of the current test line; this argument must contain the keep command with a valid cluster variable name. If the keep action is successful, the cluster variable and its new value are written to the [KEEP] section of the keep file.

7.31 tfe_KeepParameter

Prototype

```
int tfe_KeepParameter( int nParameter, char* pszValue )
```

Parameters

#	type	prototype name	description
1	int	nParameter	index number of parameter with keep command
2	char*	pszValue	value to keep

Return values

type	possible values	description
int	1	value successfully kept
	0	value could not be kept

Description

This function sets the value of a cluster variable in a specified parameter of the current action word. Its functionality is the same as that of tfe_KeepArgument, only the index specifies the action word's parameter, not the test line's argument.

7.32 tfe_ProcessNextLine

Prototype

```
int tfe_ProcessNextLine( )
```

Parameters

This function has no parameters.

Return values

type	possible values	description
int	1	next valid line with a user defined action word found
	0	no valid line with a user defined action word found

Description

This function makes the Engine start from the current test line and process all subsequent cluster lines until it either finds a valid test line with a user defined action word or no more lines at all; in the first case this test line is processed and the integer one (1) is returned, in the second case only zero (0) is returned. Lines of sub clusters and templates are taken as the subsequent lines after sub cluster and template calls; the lines of the calling cluster after such calls are again taken as the one following the last one of the sub clusters and templates.

7.33 tfe_RegisterActionWord

Prototype

```
int tfe_RegisterActionWord( char* pszActionWord,
                            char* pszActionWordFunction )
```

Parameters

#	type	prototype name	description
1	char*	pszActionWord	action word to register
2	char*	pszActionWordFunction	associated action word function

Return values

type	possible values	description
int	1	registration successful
	0	registration failed

Description
This function registers a user defined action word along with its associated action word function; this action word may not be known to the Engine already.

7.34 tfe_RegisterHeaderWord

Prototype

```
int tfe_RegisterHeaderWord( char* pszHeaderWord, char* pszReportTag )
```

Parameters

#	type	prototype name	description
1	char*	pszHeaderWord	header word to register
2	char*	pszReportTag	associated report tag

Return values

type	possible values	description
int	1	registration successful
	0	registration failed

Description
This function registers a user defined header word along with its associated tag for the report header; this action word may not be known to the Engine already.

7.35 tfe_ReportCheck

Prototype

```
int tfe_ReportCheck( char* pszDescription, char* pszExpected,
                     char* pszRecorded, int nCheckResult )
```

Parameters

#	type	prototype name	description
1	char*	pszDescription	description text to write in the report
2	char*	pszExpected	expected value to test agains
3	char*	pszRecorded	value to check
4	int	nCheckResult	result of check to write in the report

Return values

type	*possible values*	*description*
`int`	1	check logged
	0	check could not be logged

Description
This function allows a custom compare function to place its result in the report. This compare function can place the result of the compare into the report by setting nResult: 0 (check failed) or 1 (check successful)

7.36 tfe_ReportComment

Prototype
```
int tfe_ReportComment( char* pszComment )
```

Parameters

#	*type*	*prototype name*	*description*
1	`char*`	pszComment	comment text to write in the report

Return values

type	*possible values*	*description*
`int`	1	comment logged
	0	comment could not be logged

Description
This function logs a specified string, which is to be placed in the report as a comment text belonging to the current test line.

7.37 tfe_ReportError

Prototype
```
int tfe_ReportError( char* pszErrorText, int nErrorType )
```

Parameters

#	*type*	*prototype name*	*description*
1	`char*`	pszErrorText	error text to write in the report
2	`int`	nErrorType	number specifying the type of error

Return values

type	*possible values*	*description*
`int`	1	error logged
	0	error could not be logged

Description
This function logs a specified string, which is to be placed in the report as the text of an error of a specified type, belonging to the current test line. Four different error types can be specified: warning, error, check error and fatal error whose codes are 0, 1, 2 and 3 respectively.

7.38 tfe_ResetEngine

Prototype
```
int tfe_ResetEngine( )
```

Parameters
This function has no parameters.

Return values

type	*possible values*	*description*
`int`	1	Engine reset
	0	Engine could not be reset

Description
This function resets the Engine, returning it to its original state after creation, erasing all registered and processed data.

7.39 tfe_StartEngine

Prototype
```
int tfe_StartEngine( char* pszCluster, char* pszReport,
                     char* pszIniFile )
```

Parameters

#	*type*	*prototype name*	*description*
1	`char*`	pszCluster	cluster file to process
2	`char*`	pszReport	name of report in which to write the results
3	`char*`	pszIniFile	ini file containing the Engine settings

Return values

type	*possible values*	*description*
`int`	1	Engine started
	0	Engine could not be started

Description
This function makes the Engine ready to process a specified cluster. Besides the cluster file, the report file and the ini file with the settings must be specified; if cluster or report file are left empty, the values are taken from the settings; the same if their paths are not specified. If the specified ini file contains invalid values for settings, or if the license key is missing or invalid, the operation will fail. If so specified in the settings, an Engine Message Centre will be started.

7.40 tfe_StopEngine

Prototype
```
int tfe_StopEngine( )
```

Parameters
This function has no parameters.

Return values

type	*possible values*	*description*
`int`	1	Engine stopped
	0	Engine could not be stopped

Description
This function stops the Engine processing its cluster; all open cluster and log files are closed; all Engine Message Centers connected to the Engine are closed.

7.41 tfe_UnregisterActionWord

Prototype
```
int tfe_UnregisterActionWord( char* pszActionWord )
```

Parameters

#	*type*	*prototype name*	*description*
1	`char*`	pszActionWord	action word to unregister

Return values

type	*possible values*	*description*
`int`	1	action word successfully unregistered
	0	action word could not be unregistered

Description
This function removes a user defined action word from the list of registered action words; if the specified action word is not user defined, or not known to the Engine at all, the operation will fail.

7.42 tfe_UnregisterAllActionWords

Prototype
```
int tfe_UnregisterAllActionWords( )
```

Parameters
This function has no parameters.

Return values

type	possible values	description
int	>= 0	number of action words, which are unregistered

Description
This function removes all registered, user defined action words—'normal' action words and header words; all built-ins remain.

7.43 tfe_UnregisterHeaderWord

Prototype
```
int tfe_UnregisterHeaderWord( char* pszHeaderWord )
```

Parameters

#	type	prototype name	description
1	char*	pszHeaderWord	header word to unregister

Return values

type	possible values	description
int	1	header word successfully unregistered
	0	header word could not be unregistered

Description
This function removes a user defined header word from the list of registered action words; if the specified action word is not user defined, or not known to the Engine at all, the operation will fail.

7.44 Legacy functions

The interface functions described in the previous paragraph comprise the full functionality of the Engine. However, in order to be compliant with some older Engine versions, some more interface functions are supported, most of which are more or less synonyms of the ones previously discussed. Below those legacy functions' prototypes are listed in alphabetical order along with descriptions of how they map on or how they differ from the other interfaces.

```
char* eng_action( char* pszActionWord )
```

The function **eng_action** is the same as **tfe_GetActionWordFunction**, if the parameter pszActionWord is filled with an empty string; if this parameter is filled with the name of a registered action word, the action word function associated with that one is returned.

```
char* eng_argument( int nArgument )
```

The function **eng_argument** is the same as **tfe_GetArgument**.

```
int eng_argument_count( )
```

The function **eng_argument_count** is the same as **tfe_GetNumberOfArguments**.

```
int eng_check_argument(char* pszRecorded, int nArgument )
```

The function **eng_check_argument** is much the same as **tfe_CheckArgument**; however, the order of the parameters is reversed, and results of the check are printed horizontally instead of vertically in the report.

```
int eng_check_value( char* pszDescription, char* pszExpected,
                     char* pszRecorded )
```

The function **eng_check_value** is the same as **tfe_CheckString**.

```
char* eng_copyright( )
```

The function **eng_copyright** is the same as **tfe_GetCopyright**.

int eng_create_synonym(char* pszActionWord, char* pszSynonym)

The function **eng_create_synonym** is the same as **tfe_CreateActionWordSynonym**.

```
int eng_date_argument( int nIndex )
```

The function **eng_date_argument** has no meaningful implementation; it always returns one (1.) Backwards compatibility is the only reason for inclusion.

```
int eng_error( char* pszErrorText, int nErrorType )
```

The function **eng_error** is the same as **tfe_ReportError**.

```
int eng_keep_argument( char* pszValue, int nArgument )
```

The function **eng_keep_argument** is much the same as **tfe_KeepArgument**; however, the order of the parameters is reversed.

```
char* eng_legend( int nIndex )
```

The function **eng_legend** is the same as **tfe_GetArgumentDescription**.

```
int eng_line( )
```

The function **eng_line** is the same as **tfe_ProcessNextLine**.

```
int eng_line_count( )
```

The function **eng_line_count** returns the total number of lines in the main cluster file.

```
int eng_message( char* pszMessage )
```

The function **eng_message** is the same as **tfe_DisplayMessage**.

```
int eng_read_setting( char* pszDest, char* pszIniFile,
                      char* pszSection, char* pszKey )
```

The function **eng_read_setting** offers functionality, which is not really Engine functionality: it reads a setting from an ini file; pszIniFile must contain the name and path of this ini file, pszSection and pszKey the respective section and key; the read value is placed in pszDest. The return value is always one (1.)

```
int eng_register_actionword( char* pszActionword,
                             char* pszActionWordFunction )
```

The function **eng_register_actionword** is the same as **tfe_RegisterActionWord**.

```
int eng_register_headerword( char* pszHeaderWord,
                             char* pszReportTag )
```

The function **eng_register_headerword** is the same as **tfe_RegisterHeaderWord**.

```
int eng_report( char* pszLabel, char* pszMessage )
```

The function **eng_report** is a special usage of the function **tfe_ReportComment**: the parameter pszLabel, a space character and the parameter pszMessage are concatenated, in that order, and entered as the parameter of **tfe_ReportComment**; i.e. **tfe_ReportComment**(pszLabel + " " + pszMessage).

```
int eng_start( char* pszCluster, char* pszReport, char* pszIniFile )
```

The function **eng_start** is the same as **tfe_StartEngine**.

```
int eng_status( )
```

The function **eng_status** has no meaningful implementation; it always returns one (1.) Backwards compatibility is the only reason for inclusion.

```
int eng_stop( )
```

The function **eng_stop** is the same as **tfe_StopEngine** directly followed by t**fe_GenerateReport**.

```
int eng_unregister_actionword( char* pszActionWord )
```

The function **eng_unregister_actionword** is the same as **tfe_UnregisterActionWord**.

```
int eng_unregister_all_actionwords( )
```

The function **eng_unregister_all_actionwords** is the same as **tfe_UnregisterAllActionWords**.

```
char* eng_version( )
```

The function **eng_version** is the same as **tfe_GetVersion**.

```
int eng_write_setting( char* pszInifile, char* pszSection,
                       char* pszKey, char* pszValue);
```

The function **eng_write_setting** offers functionality, which is not really Engine functionality: it writes a key-value pair to a section of an ini file; pszIniFile must contain the name and path of this ini file; pszSection, pszKey and pszValue the respective section, key and value. The return value is always one (1.)