

TETware *professional*

Realtime and Embedded Systems Extension

Installation, User, Demonstration and Programmers Guide for TETware *professional* 1.4

Released: 10 January 2002

THE *Open* GROUP

The information contained within this document is subject to change without notice.

Copyright 2002 The Open Group

All rights reserved. No part of this documentation may be reproduced, stored in retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, except as stated in the end-user license agreement, without the prior permission of the copyright owners. The text of the end-user license agreement appears in Appendix A of this document. A copy of the end-user license agreement is contained in the file License, which accompanies the TETware *professional* distribution.

Motif, OSF/1, UNIX and the 'X' device are registered trademarks and TETware, IT Dial Tone and The Open Group are trademarks of The Open Group in the US and other countries.

X/Open is a trademark of The Open Group Company Limited in the UK and other countries.

Win 32TM, Windows NTTM and Windows 95TM, 98TM and 2000TM are registered trademarks of Microsoft Corporation.

This document is produced by

The Open Group
Apex Plaza
Forbury Road
Reading
Berkshire
RG1 1AX
England.

CONTENTS

1	Introduction.....	1
1.1	Preface.....	1
1.2	Audience	1
1.3	Conventions Used in this Guide.....	1
1.4	Related Documents	2
1.5	Problem Reporting	2
2	TETware <i>professional</i> Realtime Overview.....	3
2.1	Introduction.....	3
2.2	System Architecture.....	3
2.2.1	TETware <i>professional</i> Testing Model	3
2.2.2	TETware <i>professional</i> RT Testing Model	4
2.3	The TETware <i>professional</i> RT Test Manager.....	5
2.4	The TETware <i>professional</i> RT TCM and API.....	6
3	Installation Guide.....	7
3.1	Basic Installation.....	7
4	User Guide	11
4.1	Introduction.....	11
4.2	Configuration Variables	11
5	Running the Embedded Demonstration.....	15
5.1	Introduction.....	15
5.2	Selection the Test Run File	15
5.3	Using a Different Location.....	15
5.4	Selecting the IP Address	16
5.5	Running the Test Run.....	16
6	Programmers Guide.....	17
6.1	Overview	17
6.1.1	The Test Manifest	17
6.1.2	Manufacturer-specific Subsystems	17
6.1.3	Support For Tests That Modify The Process Environment	18
6.1.4	Conditional Test Purpose Processing	18
6.1.5	Test Reporting and Journaling	19
6.1.5.1	Test Case Information Lines	19

6.1.5.2	Context, Block And Sequence Numbers.....	20
6.1.6	Timeout Processing	20
6.1.7	Realtime System Resets.....	21
6.1.8	User Abort Processing	21
6.2	Test Manifest File.....	22
6.2.1	Introduction.....	22
6.2.2	File Name and Location.....	22
6.2.3	File Format.....	23
6.2.4	Test Manifest Keywords.....	23
6.2.4.1	Introduction.....	23
6.2.4.2	The <code>test-purpose</code> Keyword.....	24
6.2.4.3	The <code>test-case-instance</code> Keyword	24
6.2.4.4	The <code>tc-instance</code> Keyword	25
6.2.4.5	The <code>config-var</code> Keyword	25
6.2.4.6	The <code>infoline</code> Keyword.....	27
6.2.4.7	The <code>result</code> Keyword.....	28
6.2.4.8	The <code>timeout</code> Keyword	28
6.2.4.9	The <code>timeslices</code> Keyword.....	29
6.2.4.10	The <code>timeout-factor</code> Keyword.....	30
6.2.5	Examples.....	30
6.3	Using the Test Manager	33
6.3.1	Introduction.....	33
6.3.2	Test Manager Name and Location.....	34
6.3.3	Configuring TETware <i>professional</i> to use the Test Manager.....	34
6.4	The TETware <i>professional</i> RT C API.....	34
6.4.1	Introduction.....	34
6.4.2	C Language Binding	35
6.4.3	Supported API functions.....	36
6.4.3.1	Introduction.....	36
6.4.3.2	Test Case Structure and Management.....	36
6.4.3.3	Insulating from the Test Environment	36
6.4.3.4	Error Handling and Reporting	36
6.4.3.5	Making Journal Entries	36
6.4.3.6	Canceling Test Purposes	36
6.4.3.7	Accessing Configuration Variables	37
6.4.3.8	Generating and Executing Processes	37
6.4.3.9	Executed Process Functions.....	37
6.4.3.10	Test Case Synchronization.....	37
6.4.3.11	Remote System Information	37
6.4.3.12	Remote Process Control.....	37

6.4.3.13	Thread Functions	37
Appendix A	- TETware <i>professional</i> License	38
Appendix B	- Manufacturer-specific Subsystems	41
B.1	Introduction	41
B.2	Subsystem Descriptions	41
B.2.1	Communication Subsystem	41
B.2.2	Exec Subsystem	42
B.2.3	Reset Subsystem	42
B.3	Source File Directories	42
B.4	API Functions	43
B.5	The tet3rt.h file	43
B.6	Return Values	43
B.7	Implementation Notes	44
B.7.1	Introduction	44
B.7.2	Conditional Compilation	44
B.7.3	Error Reporting	45
B.7.4	Use of Configuration Variables	45
B.7.5	Signal Handling	46
B.7.6	Trace Debugging	47
B.8	Communication Subsystem API	47
B.8.1	Introduction	47
B.8.2	tet3rt_msgtm_open()	47
B.8.3	tet3rt_msgtm_close()	49
B.8.4	tet3rt_msgtm_rcv()	49
B.8.5	tet3rt_msgtm_snd()	51
B.8.6	tet3rt_msgrt_open()	52
B.8.7	tet3rt_msgrt_close()	53
B.8.8	tet3rt_msgrt_snd()	53
B.8.9	tet3rt_msgrt_rcv()	54
B.9	Exec Subsystem API	56
B.9.1	Introduction	56
B.9.2	tet3rt_rt_exec()	56
B.9.3	tet3rt_rt_exit()	57
B.10	Reset Subsystem API	58
B.10.1	Introduction	58
B.10.2	tet3rt_rt_reset()	58

B.11	TETware <i>professional</i> RT Functions	59
B.11.1	Introduction.....	59
B.11.2	Defined Constants.....	59
B.11.3	Error Reporting.....	60
B.11.3.1	tet3rt_mss_printf()	60
B.11.3.2	tet3rt_mss_generror().....	60
B.11.4	Trace Debugging	62
B.11.4.1	tet3rt_mss_trace()	62
B.11.4.2	tet3rt_mss_traceflag.....	62
B.11.4.3	tet3rt_mss_tdump().....	63
B.11.4.4	tet3rt_prmsser().....	64
B.11.5	Signal Handling	64
B.11.5.1	tet3rt_block_signals()	64
B.12	Example MSS Implementations.....	65
B.13	Example Socket-based Implementation	65
B.13.1	Introduction.....	65
B.13.2	Communication Subsystem	66
B.13.2.1	Subsystem Description.....	66
B.13.2.2	Subsystem-specific Configuration Variables.....	66
B.13.3	Exec Subsystem	67
B.13.3.1	Test Manager Side	67
B.13.3.2	Realtime System Side	67
B.13.4	Reset Subsystem	67
B.14	Example Serial Line Implementation.....	68
B.14.1	Introduction.....	68
B.14.2	Communication Subsystem	68
B.14.3	Exec Subsystem	68
B.14.3.1	Test Manager Side	68
B.14.3.2	Realtime System Side	68
B.14.4	Reset Subsystem	69

LIST OF FIGURES

Figure 1: Simple TETware <i>professional</i> Testing Model	4
Figure 2: Simple TETware <i>professional</i> RT Testing Model	5
Figure 3: Local: Embedded Mode	12

LIST OF TABLES

Table 1: Test Case Information Line Prefix Strings	20
----------------------------------------------------------	----

1 Introduction

1.1 Preface

This document is the Realtime and Embedded Systems Extension to the TETware *professional* User Guide.

TETware *professional* is a Test Execution Management System that takes care of the administration, reporting, and sequencing of the tests providing a single common user interface for all of the tests that you develop.

TETware *professional* has been tested and used on UNIX, Linux and Windows host operating systems.

Throughout this document, the Windows NT, 2000 and 9x operating systems are referred to collectively as **Win32** systems. The individual names are only used when it is necessary to distinguish between them.

1.2 Audience

This document is intended to be read by systems administrators who will install TETware *professional* on their computer systems, and by software testing engineers who will use TETware *professional* to run test suites.

1.3 Conventions Used in this Guide

The following typographic conventions are used throughout this guide:

- `Courier` font is used for function and program names, literals and file names. Examples and computer-generated output are also presented in this font.

- The names of variables are presented in *italic font*. You should substitute the variable's value when typing a command that contains a word in this font.
- **Bold font** is used for headings and for emphasis.

1.4 Related Documents

Refer to the following documents for additional information about TETware *professional*:

TETware *professional* Installation Guide
TETware *professional* Demonstration Guide
TETware *professional* Release Notes
TETware *professional* Help Pages
TETware Programmers Guide
TETware Installation Guide

The TETware *professional* Installation Guide contains important information about how to install and use TETware *professional*. You should read the Installation Guide thoroughly before attempting to install and use each new release of TETware *professional*.

1.5 Problem Reporting

If you have subscribed to TETware *professional* support and you encounter a problem while installing and using TETware *professional*, you can send a support request by electronic mail using the dedicated email address that is provided. Evaluators should email to tetware_manager@opengroup.org

All Problem Reports are welcome and actively encouraged. The more problems that are found and fixed the better the product will be. Please submit all bugs and queries found. Also, please submit requests for features and upgrades.

2 TETware *professional* Realtime Overview

2.1 Introduction

TETware *professional* is a test execution management system that is designed to operate on systems that support at least the functionality described in POSIX 1003.1 (1990). The POSIX standard for Embedded Realtime Systems (POSIX 1003.13) defines four profiles for realtime systems, three of which do not include all the functionality described in P1003.1. Therefore, TETware *professional* cannot be used to execute test cases directly on these systems.

TETware *professional* Realtime (TETware *professional* RT) is an extension to TETware *professional*, which enables TETware *professional* to control the execution of tests on Embedded POSIX Realtime Systems that cannot support TETware *professional* directly.

Test cases that execute on the realtime system are linked with the TETware *professional* RT version of the C Test Case Manager (TCM) module and API library. The TETware *professional* RT C API supports a substantial subset of the functions available in the Lite version of the TETware *professional* C API.

2.2 System Architecture

2.2.1 TETware *professional* Testing Model

In a “normal” non-distributed testing setup, both TETware *professional*, and the test cases that it processes, run on the same system. The whole process is driven by a list of test cases contained in the scenario file. A Test Case Manager module is linked into each test case executable. The TCM calls each Test Purpose (TP) function in turn. Each TP completes whatever processing is necessary to perform the test, and then calls an API function to record a result. When all the TP functions have been called, the Test Case Manager exits. Finally, TETware *professional* gathers the results of each TP, writes them to the journal and moves on to the next test case.

Figure 1 provides a simple illustration of the relationship between the main TETware *professional* components in the “normal” non-distributed testing model.

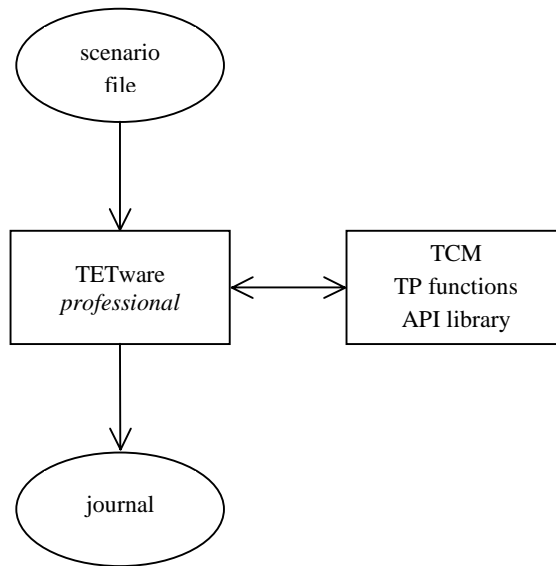


Figure 1: Simple TETware *professional* Testing Model

2.2.2 TETware *professional* RT Testing Model

When testing embedded realtime systems, this model needs to be modified. This is mainly for the following reasons:

- TETware *professional* cannot run on the realtime system. All the control operations must be performed on a host system.
- Operating system facilities on the realtime system may be limited. If a test case malfunctions on the realtime system, it may be necessary to reset the system in order to regain control.

The required modification is accomplished by using TETware *professional*'s **exec tool** facility to run the TETware *professional* RT Test Manager on the host system (that is: the system on which TETware *professional* runs). The Test Manager acts as an agent for the test case that is running on the realtime system.

Figure 2 provides a simple illustration of the relationship between the main components in the TETware *professional* RT testing model.

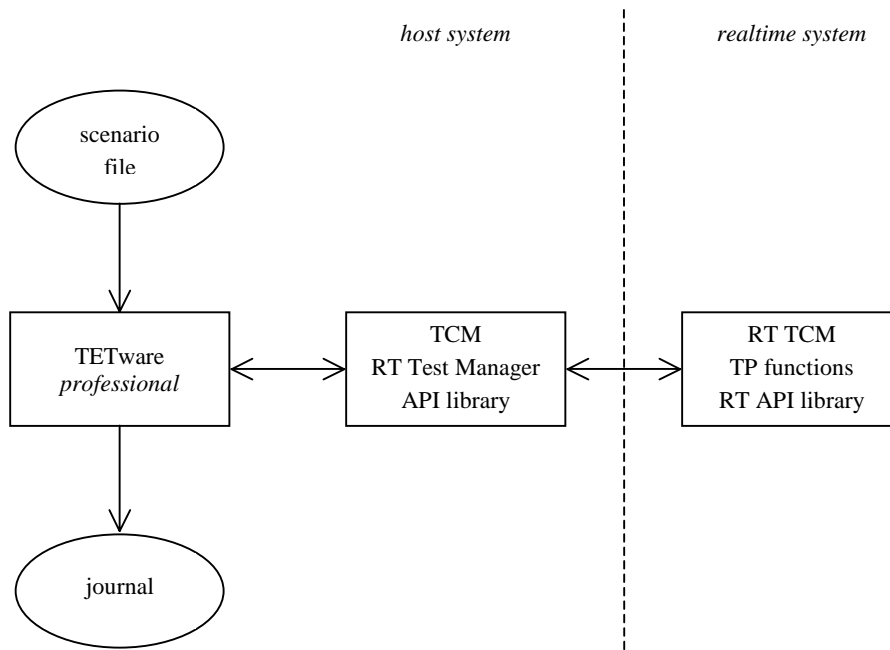


Figure 2: Simple TETware *professional* RT Testing Model

2.3 The TETware *professional* RT Test Manager

TETware *professional* executes a new instance of the Test Manager each time it executes a test case. The Test Manager performs the following operations:

1. Read information from the test manifest, including information about the arrangement of ICs and TP functions in the test case.
2. Use the dynamic test case interface to adapt itself to the IC/TP arrangement described in the test manifest.
3. Load the test case onto to realtime system and execute it.
4. Open a communication channel to the test case on the realtime system.
5. Instruct the TCM on the realtime system to invoke the test case's startup function, TP functions and cleanup function.
6. For each of these functions, enter a service loop, responding to requests from the TCM/API on the realtime system. The loop is terminated when the function returns to the realtime system's TCM, or when a timeout expires.

7. Deliver a TP function's result to the journal.
8. If the TP timed out: Reset the realtime system.

Thus the Test Manager provides the interface between TETware *professional* running on the host system, and the test case running on the realtime system. From TETware *professional*'s point of view the Test Manager looks like an API-conforming test case.

2.4 The TETware *professional* RT TCM and API

Each test case that is to run on the realtime system is linked with the TETware *professional* RT versions of the C TCM and API library. As in TETware *professional*, both single-threaded and thread-safe versions of these components are supplied. A substantial subset of the API functions implemented in TETware *professional*-Lite is available in the TETware *professional* RT version of the API library. Further details are presented in Section 6.4 "The TETware *professional* RT C API".

Although the supported API functions are the same, in many cases the implementations are quite different. For example, functions that write information to the execution results file in TETware *professional* instead send the information to the Test Manager in TETware *professional* RT. When the Test Manager receives this information, it writes the information to the execution results file on the host system.

3 Installation Guide

3.1 Basic Installation

This chapter describes how to build and install the Embedded Module on TETware *professional*.

The Embedded Module is designed to be built and installed on top of an existing TETware *professional* application.

See the TETware *professional* Installation Guide for instructions on how to extract and install a copy of TETware *professional*.

The Embedded Module should be loaded into the same directory that you have extracted TETware *professional*, and in which the `TETware_professional.jar` exists.

```
tar xvf embedded.tar
```

Change the directory to the source directory

```
cd src
```

Create a native “`defines.mk`” file in this directory. Examples of different `defines.mk` files from a variety of different operating systems are available in the directory “`src/defines`”. Choose the one most like your operating system from the `defines` directory, copy it into the `src` directory and rename it “`defines.mk`”. For example

```
cp defines/solaris7.mk defines.mk
```

If an example does not exist for your operating system, instructions for creating a new `defines.mk` are contained in the TETware Installation Guide.

Change the directory to the embedded system source directory.

```
cd tet3rt
```

Run the script to prepare the installation

```
sh install.sh
```

Create a remote system “defines_rtsys.mk” file in this directory. Examples of this file are given in this directory.

This file is similar to the defines.mk file mentioned above but relates to the compilation instructions of the realtime embedded system. Copy the example you wish to use to “defines_rtsys.mk”. For example:

```
cp defines_rtsys_solaris7.mk defines_rtsys.mk
```

If none of the examples are suitable then edit the template version “defines_rtsys_template.mk” with the suitable information, again using the TETware Installation Guide for instructions as to how to do this.

Change directory to the source directory for the embedded system interface code

```
cd msslib_rtsys
```

This directory `msslib_rtsys` contains Manufacture-specific functions for use by TETware *professional* RT on the embedded system.

The distribution contains two example implementations of these functions in the subdirectories `sockets_example` and `serial_example`. You should copy the source files from one of these examples into the `msslib_rtsys` directory. If necessary these examples can be modified.

If they are not suitable you will need to implement versions of these functions for each Realtime System that is to be used in conjunction with the TETware *professional* RT Test Manager.

The following functions must be provided on the RT-system:

```
tet3rt_msgrt_open()
tet3rt_msgrt_close()
tet3rt_msgrt_send()
tet3rt_msgrt_recv()
tet3rt_rt_exit()
```

Details of how to write other interfaces are contained in Appendix C “Manufacturer-specific subsystems”.

Change directory to the source directory for the native system interface code

```
cd ../msslib_native
```

The directory `msslib_native` contains Manufacture-specific functions for use by the native system.

The distribution contains two example implementations of these functions in the subdirectories `sockets_example` and `serial_example`. You should copy the source files from one of these examples into the `msslib_native` directory. If necessary who may edit this source code for your implementation.

You may need to implement your own versions of these functions for each Realtime System that is to be used in conjunction with the Embedded Module.

The following functions must be provided on the host system (that is: the system on which the Test Manager runs):

```
tet3rt_msgtm_open()
tet3rt_msgtm_close()
tet3rt_msgtm_send()
tet3rt_msgtm_recv()
tet3rt_rt_exec()
tet3rt_rt_reset()
```

Details of how to write other interfaces are contained in Appendix C “Manufacturer-specific subsystems”.

Return to the Embedded Source directory

```
cd ..
```

Compile the Source

```
make install
```

Embedded TETware *professional* is now ready to use.

4 User Guide

4.1 Introduction

The Local: Embedded tab in the TETware *professional* GUI (see Figure 3) allows users to specify variables that TETware *professional* uses in execute mode to determine how to process a test case on an Embedded Realtime System.

4.2 Configuration Variables

The configuration variables are used in this mode are:

TET3RT_RTSYS_ID Specifies the string that the Test Manager passes to Manufacturer-specific functions that take an `rtsys_id` argument. Whether or not the MSS actually uses it depends on the implementation. This variable **must** be specified.

TET3RT_TRACE_MSS Control the generation of trace messages by MSS functions in the Test Manager. Whether or not trace messages are actually generated depends on the MSS implementation. Typically this variable is set to a value between 0 and 10. The use of this variable is optional; if it is not specified, its value defaults to zero (no trace messages generated).

TET3RT_MSS_RTSYS_TRACE Controls the generation of trace messages by Manufacturer-specific subsystems (MSS) functions on the realtime system. Whether or not trace messages are actually generated depends on the MSS implementation. Typically this variable is set to a value between 0 and 10. The use of this variable is optional; if it is not specified, its value defaults to zero (no trace messages generated).

TET3RT_MSS_TRACE2JNL Specifies whether or not MSS trace message generated by the Test Manager should be printed in the journal as well as appearing on the standard error stream. Whether or not trace message are actually generated depends on the MSS implementation. The possible values are `True` and `False`. The use of this variable is optional; if not specified; its value defaults to `False`.

TET3RT_TP_TIMEOUT Specifies the number of seconds to be used as the default test purpose timeout. The actual timeout that the Test Manager applies to a test purpose depends on information in the test manifest as well as on the value of this variable; see Section 6.1.8 "Timeout Processing" for further details. This variable **must** be specified.

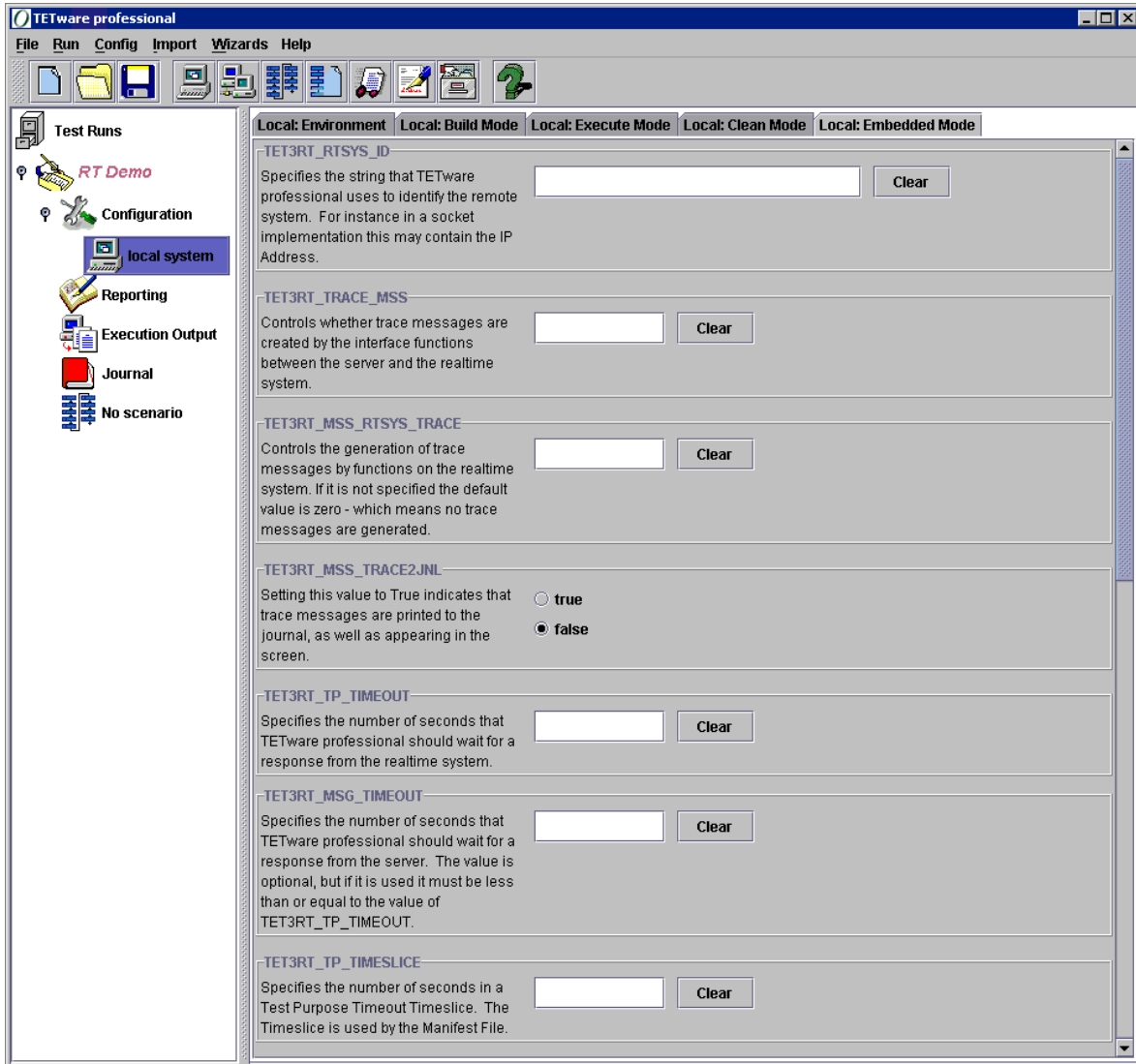


Figure 3: Local: Embedded Mode

TET3RT_MSG_TIMEOUT Specifies the number of seconds that TETware *professional* should wait for the TCM to respond to a request when in server mode (that is: when the TCM is waiting for the next instruction from the Test Manager). If specified, the value of this variable should be greater than 1 and less than or equal to the value of TET3RT_TP_TIMEOUT. The use of this variable is optional; if it is not specified, its value defaults to that specified by TET3RT_TP_TIMEOUT.

TET3RT_TP_TIMESLICE Specifies the number of seconds in a timeslice that is used by the Test Manager when calculating the timeout to be applied to a test purpose. The number of timeslice to be applied to a test purpose may be specified in the test manifest. See Section 6.1.8 "Timeout Processing" for further details. This variable **must** be specified.

5 Running the Embedded Demonstration

5.1 Introduction

This chapter describes how to run the Embedded Demonstration that is supplied with TETware *professional* RT.

The demonstration is designed to be used on the sockets implementation of TETware *professional* RT.

First follow the instructions contained within the Installation Guide of the Realtime and Embedded Systems Extension.

It is recommended that you install TETware *professional* and the Embedded Module into `/usr/local` for this demonstration, but if this is not possible, instructions are given on how to change the Embedded Demonstration to use a different location.

5.2 Selection the Test Run File

Once the installation of the Embedded Module is complete, click on the Open Test Run icon and select `embedded_demo.trf`.

5.3 Using a Different Location

This Test Run File presumes that TETware *professional* has been installed into `/usr/local`. If this is not the case various options will need to be changed.

In Configuration: Local System: Local Environment:

- `TET_ROOT` should point to the location of the `tetware_professional.jar`
- `TET_SUITE_ROOT` should point to the location of the `embedded_demo` directory

In Configuration: Local System: Local Execute:

- `TET_EXEC_TOOL` should point to the location of the binary “`tet3rtm`”.


The scenario file will also need to be opened. This can be done by using the menu `Config: Add Scenario File`. The scenario file can be found in the `embedded_demo` directory and is called `tet_scen`.

5.4 Selecting the IP Address

The IP address of the embedded system also needs to be set. This is held in the `Configuration: Local System: Local Embedded: TET3RT_RTSYS_ID`. Currently this is set to the loop back address in `127.0.0.1`.

5.5 Running the Test Run

Once the Test Run has been set up, it can be run by any of the following methods:

- Clicking on the Execute Test Run icon on the tool bar The icon is a document with a red play button symbol at the bottom right.
- Using the Keyboard Shortcut `Alt-E`
- Selecting the menu option: `Run: Execute Test Run`

6 Programmers Guide

6.1 Overview

6.1.1 The Test Manifest

It will be apparent from the description presented in the Section 2 that the functions that would be performed by the TCM module in TETware *professional* are shared in TETware *professional* RT between the RT Test Manager on the host system and the RT TCM on the realtime system. Thus, some of the information about the test case that is available to the TETware *professional* TCM cannot be accessed by the TETware *professional* RT Test Manager. In particular, the Test Manager cannot easily access information about the arrangement of Invocable Components (ICs) and Test Purpose functions (TPs) that is defined in the test case's `test_list[]` array.

Instead, the TETware *professional* RT Test Manager reads this (and other) information from a test manifest file soon after it starts up. This is an additional data file used by TETware *professional* RT, which must be provided by the test suite author. Each test case must have a test manifest file associated with it.

6.1.2 Manufacturer-specific Subsystems

In order to enable the Test Manager to work with the TCM/API running on any particular realtime system, it is necessary to customize both these components. This is achieved by the use of Manufacturer-specific subsystems (MSS). Each subsystem is responsible for performing specific tasks in connection with the realtime system to which it applies. These subsystems must be implemented by, or on behalf of, the manufacturer of the realtime system that is to be tested.

The following subsystems are defined:

- Communication subsystem (implemented on host system and on realtime system).
Communicate between the host system and the realtime system.
- Execution subsystem (implemented on host system and on realtime system).
Transfer a test case from the host system to the realtime system and execute it.
Exit from (or terminate) a process running on the realtime system.
- Reset subsystem (implemented on host system).
Reset the realtime system.

The interface to each of these subsystems is provided by one or more functions (the MSS API functions). These functions are described in Appendix C “Manufacturer-specific subsystems”.

When the Test Manager or TETware *professional* RT TCM/API needs to make use of the services provided by one of the subsystems, it calls the appropriate MSS API function.

6.1.3 Support For Tests That Modify The Process Environment

In a verification test suite, many test methods need particular environmental conditions to be set up, or may modify their process environment in some way, that might have an adverse effect on the behavior of subsequent tests in the test case. When writing such tests, a common strategy employed by test authors to overcome this problem is to put one or more test purpose functions in a child process or subprogram started by a call to `tet_fork()`.

Systems that conform to POSIX realtime profiles 51 and 52 are only required to support a single process so, on those systems, `tet_fork()` cannot be implemented; thus test strategies of this kind cannot be used.

In order to overcome this problem, the Test Manager supports the concept of multiple **test case execution instances**. A keyword in the test manifest indicates when a new test case instance should be started.

Normally, when the Test Manager executes a test case on the realtime system, it instructs the test case to invoke the startup, test purpose and cleanup functions in the same way as would TETware *professional*. However, when the test suite author indicates in the test manifest that a new test case instance should be started part way through the list of TP functions, the Test Manager performs the following operations:

1. Instruct the test case to invoke the cleanup function;
2. Instruct the test case to exit;
3. Load and execute a new instance of the test case on the realtime system;
4. Instruct the test case to invoke the startup function;
5. Instruct the test case to invoke the next TP function.

6.1.4 Conditional Test Purpose Processing

In a typical test suite there may be many test purpose functions that use the value of a configuration variable to determine whether or not a test should be performed. If the variable's value indicates that the test should not be performed, the test purpose function simply emits a test case information line, which says why the test cannot be performed, and registers a result such as UNSUPPORTED or UNTESTED.

When testing realtime systems, it would be inefficient to go to the trouble of executing one or more test case instances on a realtime system just to check the value of configuration variables. In order to make test suite execution more efficient, it is possible to instruct the Test Manager whether or not to invoke a test purpose function depending on the value(s) of one or more variables in the execute mode configuration. This is done by using keywords in the test manifest. These keywords are described in Section 6.2.4 “Test Manifest Keywords”.

In the test manifest, each test purpose may have one or more **configuration variable expressions** associated with it. Each of these expressions has zero or more test case information lines, and exactly one result code, associated with it. Before the Test Manager instructs the realtime system to invoke a test purpose function, it evaluates each expression in turn. If an expression is `TRUE`, the Test Manager prints the test case information line(s) to the journal, generates the result and steps on to the next test purpose. Otherwise, if the expression is `FALSE`, the Test Manager steps on to the next expression. If none of the expressions are `TRUE`, the default action is to instruct the realtime system to invoke the test purpose function in the normal way.

6.1.5 Test Reporting and Journaling

6.1.5.1 Test Case Information Lines

When a test case running on the realtime system makes a call to one of the functions that writes to the journal¹, the API on the realtime system sends the data specified in the call to the Test Manager on the host system. Thus it is the Test Manager that makes the entry in the execution results file. In addition, the TCM/API on the realtime system may send Test Case Manager Messages to the Test Manager, and the Test Manager itself may write Test Case Information lines to the execution results file.

In order to enable users and report writers to distinguish between Test Case Information lines that originate from different sources, the Test Manager prepends strings to the different types of line, as shown in the following table:

¹ These functions are: `tet_infoline()`, `tet_minfoline()`, `tet_printf()`, `tet_vprintf()` and `tet_result()`.

Source of information line	Prefix string
Test Manager	TM :
Manufacturer-specific subsystem	MSS :
Trace message from MSS	MSS_TRACE :
RT-system TCM message	RTSYS_TCM
Test case	(none)

Table 1: Test Case Information Line Prefix Strings

6.1.5.2 Context, Block And Sequence Numbers

It will be seen from the information presented in the previous section that Test Case Information lines that appear in the journal may originate from a number of sources, and not just from the TCM/API that runs on the realtime system. Therefore it is not really possible for journal context and block numbers to have the same meaning in TETware *professional* RT as they do in TETware *professional*. Instead, in TETware *professional* RT, these values are determined as follows:

Context

All Test Case Information lines are written to the execution results file with the same context value. This value is derived from the Test Manager's process ID in the usual way. A call to `tet_context()` on the realtime system has no effect on context value that is maintained by the Test Manager's TCM/API.

Block

A call to `tet_block()` on the realtime system causes the API to instruct the Test Manager to call `tet_block()`. Thus the block number in the execution results file will change at certain times but each change will affect all lines written after a call to `tet_block()` in any process or thread that is running on the realtime system.

Sequence

This is calculated by the Test Manager's TCM/API in the normal way. The value of the sequence counter maintained by the TCM/API on the realtime system is not visible in the execution results file.

6.1.6 Timeout Processing

The Test Manager invokes each test purpose function under the control of a timeout. If a test purpose is still running when the timeout expires, the Test Manager resets the realtime system, thus terminating the test purpose. Then, if there are more test purpose functions to invoke, the Test Manager starts another test case instance on the realtime system in which to invoke subsequent test purpose functions.

The Test Manager uses information contained in the execute mode configuration and in the test manifest in order to determine the timeout to be applied to each test purpose function. This information is used as follows:

- A default test purpose timeout is defined in the execute mode configuration.
- The default timeout may be overridden by a longer test purpose timeout defined in the test manifest. This timeout may be defined at the test purpose or the test case level; in the latter case the timeout applies to all the test purposes in the test case unless a longer timeout is defined for a particular test purpose.

It should be noticed that a more specific timeout is only used if it is longer than the less specific one. Thus, a per-test-purpose value of 5 seconds does not override a default value of 10 seconds.

In addition to the absolute timeout value just described, it is possible for a relative timeout to be specified in the test manifest in terms of a number of time slices, where the length of one time slice is defined in the execute mode configuration. Once again, the number of time slices may be specified at the test purpose or test case level. If the length of a time slice as defined in the execution mode configuration is related to the speed of the realtime system in some way, it is possible for a test suite author to use this mechanism to relate a timeout to that speed.

The Test Manager calculates both the absolute timeout and, when the test manifest contains a number of time slices, the relative timeout as well. The actual timeout applied to a test purpose is the greater of the two values thus calculated.

6.1.7 Realtime System Resets

The Test Manager may reset the realtime system under the following circumstances:

- when a test purpose times out; or
- when it receives a User Abort notification from TETware *professional*.

The Test Manager calls an MSS function to perform this task. There are two levels of reset: soft and hard. The Test Manager requests a hard reset when the MSS function indicates that a previous soft reset operation has failed.

6.1.8 User Abort Processing

It is possible for TETware *professional* to deliver a SIGTERM signal to the Test Manager. Usually this occurs when TETware *professional* wants to interrupt test case processing after it receives a keyboard signal.

When the Test Manager receives a SIGTERM, it instructs the test case to terminate, then waits for a short time to receive an exit notification from the test case. If this notification is not received within that time, the Test Manager resets the realtime system.

6.2 Test Manifest File

6.2.1 Introduction

When the Test Manager executes a test case on the realtime system, it needs to know certain information about the test case. The test suite author makes this information available to the Test Manager by providing a **test manifest file** with each test case. The test manifest file contains the following information:

- Arrangement of invocable components and test purpose functions.
- Assignment of test purpose functions to test case execution instances.
- Timeout parameters.
- Configuration variable expressions, test case information lines and result codes.

6.2.2 File Name and Location

The name of the test manifest file is derived from the test case name, as follows:

- if the last component of the name of the test case starts with T., the name of the test manifest file is constructed by replacing the T. prefix with an M. prefix²;
- otherwise: the name of the test manifest file is constructed by prepending an M. prefix to the last part of the test case name.

For example: if the name of a test case is T.open, the name of the corresponding test manifest file is M.open. Alternatively, if the name of a test case is tc1, the name of the corresponding test manifest file is M.tc1.

The Test Manager looks for the test manifest file in the test case execution directory. Therefore, if an alternate execution directory is being used, the build process must copy the test manifest file to its place in the alternate execution directory hierarchy at the same time that it copies the test case executable and any other required files.

² This arrangement works well with the test case naming convention that is used in many of The Open Group's test suites.

6.2.3 File Format

The test manifest file is a plain text file. Each non-blank, non-comment line starts with a keyword and a colon. When a keyword takes arguments, the colon is followed by one or more spaces and/or tabs, then the arguments themselves. Blank lines and comment lines starting with # are ignored.

6.2.4 Test Manifest Keywords

6.2.4.1 Introduction

The keywords that may appear in a test manifest file are:

```
config-var
infoline
result
tc-instance
test-case-instance
test-purpose
timeout-factor
timeout
timeslices
```

Ordering of keywords is significant. For example, if the `timeout` keyword appears before the first `test-purpose` keyword, it supplies a timeout value to be used by every test purpose in the test case. By contrast, if a `timeout` keyword appears after a `test-purpose` keyword, it supplies a timeout value to be used only by the current test purpose.

These keywords are described in more detail in the subsections that follow.

6.2.4.2 The `test-purpose` Keyword

Synopsis

```
test-purpose: icnum
```

Description

This keyword describes a test purpose function. *icnum* defines the Invocable Component to which this test purpose belongs.

The number of `test-purpose` keywords and their associated IC numbers in the test manifest file must exactly match the number of test purpose functions and their associated IC numbers that are defined in the test case's `tet_testlist[]` array.

6.2.4.3 The `test-case-instance` Keyword

Synopsis

```
test-case-instance:
```

Description

When this keyword appears it causes the Test Manager to end the currently running test case instance and start a new test case instance before issuing the next instruction to invoke a test purpose function. The reason for providing this functionality is described in Section 6.1.3 “Support For Tests That Modify The Process Environment”.

The `test-case-instance` keyword may appear in the test manifest immediately before a second or subsequent `test-purpose` keyword. Use of this keyword is optional; when no `test-case-instance` keywords appear in a test manifest, all the test purposes that are to be invoked are invoked in a single instance of the test case.

6.2.4.4 The `tc-instance` Keyword

Synopsis

```
tc-instance:
```

Description

This keyword is a synonym for `test-case-instance`.

6.2.4.5 The `config-var` Keyword

Synopsis

```
config-var: configuration-variable-expression
[
  infoline: test-case-information-line
  [...]
]
result: result-name
```

Description

This keyword defines a configuration variable expression. When it appears, it is followed by zero or more `infoline` keywords and exactly one `result` keyword. One or more of these keyword groups may appear after a `test-purpose` keyword. Use of these keyword groups is optional; when no such groups appear, the Test Manager simply invokes the current test purpose function.

When the Test Manager is about to process a test purpose function that has one or more of these keyword groups associated with it, the Test Manager first evaluates the specified *configuration-variable-expression*. If the expression is TRUE, the Test Manager prints each *test-case-information-line* to the execution results file, reports the result associated with *result-name* and steps on to the next test purpose. If the expression is FALSE, the Test Manager steps on to the next `config-var` keyword group, if there is one. Finally, if none of the configuration variable expressions are TRUE, the Test Manager instructs the test case to invoke the test purpose function.

Each *configuration-variable-expression* is written using syntax reminiscent of that used in awk. The following simple expressions are understood:

`defined(variable)` If the named *variable* is defined in the execute mode configuration, the expression is TRUE; otherwise the expression is FALSE.

`variable == "string"` The value of the named *variable* is looked up in the execute mode configuration. If *variable* is defined, its value is compared to the specified *string*. The expression is TRUE if *variable* is defined and the comparison succeeds; otherwise the expression is FALSE.

`variable != "string"` The value of the named *variable* is looked up in the execute mode configuration. If *variable* is defined, its value is compared to the specified *string*. The expression is TRUE if *variable* is not defined or if the comparison fails; otherwise the expression is FALSE.

`variable == TRUE`
`variable != FALSE` The value of the named *variable* is looked up in the execute mode configuration. If *variable* is defined, the first letter of its value is examined. The expression is TRUE if *variable* is defined and the first letter of its value is either T or t; otherwise the expression is FALSE.

`variable != TRUE`
`variable == FALSE` The value of the named *variable* is looked up in the execute mode configuration. If *variable* is defined, the first letter of its value is examined. The expression is TRUE if *variable* is not defined, or if the first letter of its value is neither T nor t; otherwise the expression is FALSE.

`variable ~ /regular-expression/` The value of the named *variable* is looked up in the execute mode configuration. If *variable* is defined, its value is matched against the specified extended *regular-expression*. The expression is TRUE if *variable* is defined and the match succeeds; otherwise the expression is FALSE.

variable !~ /regular-expression/

The value of the named *variable* is looked up in the execute mode configuration. If *variable* is defined, its value is matched against the specified extended *regular-expression*. The expression is TRUE if *variable* is not defined or if the match fails; otherwise the expression is FALSE.

These simple expressions may be combined in the usual way as follows:

! <i>expr</i>	The expression is TRUE if <i>expr</i> is FALSE.
<i>expr</i> ₁ && <i>expr</i> ₂	The expression is TRUE if both <i>expr</i> ₁ and <i>expr</i> ₂ are TRUE.
<i>expr</i> ₁ <i>expr</i> ₂	The expression is TRUE if either <i>expr</i> ₁ or <i>expr</i> ₂ is TRUE.
(<i>expr</i>)	Parentheses for grouping.

6.2.4.6 The **infoline** Keyword

Synopsis

infoline: *test-case-information-line*

Description

This keyword defines a test case information line.

When this keyword appears after a `test-purpose` keyword and before a `config-var` keyword, the Test Manager always prints the specified *test-case-information-line* to the execution result file. When this keyword appears after a `config-var` keyword, the Test Manager only prints the specified *test-case-information-line* when the corresponding configuration variable expression is TRUE.

Sequences of `infoline` keywords may be used to print multiple test case information lines. Use of this keyword is optional.

6.2.4.7 The **result** Keyword

Synopsis

```
result: result-name
```

Description

This keyword instructs the Test Manager to generate a result on behalf of a test purpose. When this keyword appears after a `test-purpose` keyword and before a `config-var` keyword, the Test Manager always generates the result associated with *result-name*.

When this keyword appears after a `config-var` keyword, the Test Manager only generates the result associated with *result-name* when the corresponding configuration variable expression is TRUE.

When this keyword is used before a `config-var` keyword, only one instance may appear. Use of this keyword before a `config-var` is optional. Exactly one instance of this keyword must appear after each `config-var` keyword and any associated `infoline` keywords.

result-name may be any of the result names that are known to the TCM³, or it may be the special name TEST-RESULT. In the latter case the Test Manager does not immediately record a result but instead instructs the realtime system to invoke the test purpose function and records the result from that.

6.2.4.8 The **timeout** Keyword

Synopsis

```
timeout: seconds
```

³ These are the names defined for the standard result codes as well as those specified with any user-defined result codes. For further details, refer to Section 4.1.3 “Custom Result Code Tab” in the TETware *professional* User Guide.

Description

This keyword specifies a test purpose timeout in seconds to be used in place of the default value specified in the execute mode configuration.

When this keyword appears before the first `test-purpose` keyword, it applies to all the test purpose functions in the test case. When this keyword appears after the first `test-purpose` keyword, it applies only to the current test purpose.

Use of this keyword is optional.

The way in which the Test Manager calculates the actual timeout to be applied to a test purpose function depends on several factors and is described in Section 6.1.8 “Timeout Processing”.

6.2.4.9 The `timeslices` Keyword**Synopsis**

```
timeslices: number
```

Description

This keyword specifies a test purpose timeout in terms of a number of timeslices. The length of a timeslice is defined in the execute mode configuration.

When this keyword appears before the first `test-purpose` keyword, it applies to all the test purpose functions in the test case. When this keyword appears after the first `test-purpose` keyword, it applies only to the current test purpose.

Use of this keyword is optional.

The way in which the Test Manager calculates the actual timeout to be applied to a test purpose function depends on several factors and is described in Section 6.1.8 “Timeout Processing”.

6.2.4.10 The `timeout-factor` Keyword

Synopsis

```
timeout-factor: number
```

Description

This keyword is a synonym for the `timeslices` keyword.

6.2.5 Examples

Example 1

```
test-purpose: 1
test-purpose: 2
test-purpose: 3
test-purpose: 4
```

This manifest accompanies a test case that has four test purpose functions, each in its own IC. It corresponds to the following definition in the test case source file:

```
static void tp1(), tp2(), tp3(), tp4();
struct tet_testlist tet_testlist[] = {
    { tp1, 1 },
    { tp2, 2 },
    { tp3, 3 },
    { tp4, 4 },
    { NULL, 0 }
};
```

Example 2

```
test-purpose: 1
test-purpose: 2
test-purpose: 2
```

This manifest accompanies a test case that has three test purpose functions; one in IC 1 and two in IC 2. It corresponds to the following definition in the test case source file:

```
static void tp1(), tp2(), tp3();
struct tet_testlist tet_testlist[] = {
    { tp1, 1 },
    { tp2, 2 },
    { tp3, 2 },
    { NULL, 0 }
};
```

Example 3

```
test-purpose: 1
test-case-instance:
test-purpose: 2
test-purpose: 3
```

In this example the first test purpose function is invoked in one test case instance and the remaining test purpose functions are invoked in another test case instance. Presumably this is because the strategy employed by the first test purpose would have some adverse effect on the behavior of subsequent test purpose functions.

Example 4

```
test-purpose: 1
info:line: this test purpose is not in use
result: NOTINUSE
```

In this example the test purpose function is not invoked on the realtime system. Instead the Test Manager prints the specified test case information line to the execution results file and records a result of NOTINUSE.

Example 5

```

test-purpose:      1
config-var:        !defined(VSX_BLOCK_DEV) || VSX_BLOCK_DEV == ""
info:line:         VSX_BLOCK_DEV is not specified
result:           UNRESOLVED
config-var:        VSX_BLOCK_DEV == "unsup"
info:line:         block devices are not supported
result:           UNSUPPORTED

```

This example shows how the value of a configuration variable may be used to control whether or not a test purpose function is invoked. There are two configuration variable expressions; each expression has a test case information line and a result associated with it.

When the Test Manager is about to invoke the test purpose function, it evaluates the first configuration variable expression. This expression ensures that `VSX_BLOCK_DEV` is defined with a non-empty value in the execute mode configuration. If the expression is TRUE, the Test Manager prints the first test case information line and reports a result of UNRESOLVED, then moves on to the next test purpose function. Otherwise, the Test Manager evaluates the second configuration variable expression. This expression checks to see if `VSX_BLOCK_DEV` has been set to `unsup`. If this expression is TRUE, the Test Manager prints the second test case information line and reports a result of UNSUPPORTED, then moves on to the next test purpose function. Otherwise, the Test Manager instructs the realtime system to invoke the test purpose function.

Example 6

```

timeout: 20
test-purpose: 1
test-purpose: 2

```

This example specifies a timeout of 20 seconds for each test purpose in the test case. (The actual timeout value used by the Test Manager is the greater of this value and the default value specified in the execute mode configuration.)

Example 7

```

test-purpose: 1
timeout: 20
test-purpose: 2

```


This example specifies a timeout of 20 seconds for the first test purpose only. (The actual timeout value used by the Test Manager is the greater of this value and the default value specified in the execute mode configuration.)

Example 8

```
test-purpose: 1
timeslices: 10
test-purpose: 2
```

This example specifies a timeout of 10 timeslices for the first test purpose only. The length of a timeslice is specified in the execute mode configuration. (The actual timeout value used by the Test Manager is the greater of this value and the default value specified in the execute mode configuration.)

6.3 Using the Test Manager

6.3.1 Introduction

This section describes how to use the TETware *professional* RT Test Manager to control the execution of test cases on a realtime system.

Before you can use the Test Manager in conjunction with a particular realtime system, both it and the TCM/API must have been built for use with that system.

The Test Manager must have been linked with the Manufacturer-specific subsystem library for the realtime system that you want to use. Likewise, each test case must have been linked with a version of the appropriate TCM/API that contains the Manufacturer-specific code for the realtime system that you want to use. See Appendix C for details on how to implement Manufacturer-specific subsystems for a realtime system.

6.3.2 Test Manager Name and Location

The Test Manager is launched by a shell script called `tet3rttm`. This script sets some environment variables, then executes the Test Manager program itself. The name of this program is `rttmprog`. Both the script and the program reside in `tet-root/bin`.

Note that the Test Manager must always be invoked from the `tet3rttm` shell script and never directly as `rttmprog`.

6.3.3 Configuring TETware *professional* to use the Test Manager

From TETware *professional*'s point of view, the Test Manager is an exec tool. Therefore, to instruct TETware *professional* to run test cases under the control of the Test Manager, you must set `TET_EXEC_TOOL` variable in the execute mode configuration to the path name of the Test Manager launcher. For reasons of portability it is best to use `tcc`'s configuration variable expansion capability to do this⁴.

For example:

```
TET_EXPAND_CONF_VARS=true
TET_EXEC_TOOL=${TET_ROOT}/bin/tet3rttm
```

6.4 The TETware *professional* RT C API

6.4.1 Introduction

The TETware *professional* RT C API is derived from the Lite version of the TET3 C API. This section lists the functions that are implemented in the TETware *professional* RT version of the C API, together with any differences between this version and the TET3 version.

Functions in this API may be used by test cases that run on a realtime system. As in TET3, both single-threaded and thread-safe versions of the API are provided. However, shared library versions

⁴ Refer to Section 5.6 “Configuration variables which modify TETware’s operation” and Section 5.8 “Configuration variable expansion” in the TETware Programmers Guide for further details.

of the API files are not provided in TETware *professional* RT. The thread-safe version of the TETware *professional* RT API supports POSIX threads.

Support for API-conforming executed subprograms⁵ can only be provided in a portable manner on a profile 54 system⁶. Since such a system is thus capable of supporting TET3 in its own right, support for executed subprograms is not provided in TETware *professional* RT.

For details of the API functions themselves, please refer to the Chapter 8 “The C API” and Chapter 10 “The Thread-safe C and C++ APIs”, both in the TETware Programmers Guide.

6.4.2 C Language Binding

Test cases that use this API are compiled on the host system using a cross-compiler and other cross-tools that are suitable for compiling programs for use on the runtime system. To the extent possible, the way in which test cases are compiled is similar to that used in the standard (non-realtime) version of TET3. In particular, the file names used by the API are the same.

Test cases written to use this API attach themselves to it through the following files:

- `tet-root/lib/tet3/tcm.o` is the single-threaded version of the TCM.
- `tet-root/lib/tet3/libapi.a` is the single-threaded version of the API library.
- `tet-root/lib/tet3/thrtcm.o` is the thread-safe version of the TCM.
- `tet-root/lib/tet3/libthrap.a` is the thread-safe version of the API library.
- `tet-root/inc/tet3/tet_api.h` contains prototypes for the functions, declarations of all the global variables, and definitions of all the structures and manifest constants that constitute the C API.

Note that the suffixes of the names of object and library files shown above are those that are used on a UNIX system. It is possible that different suffixes will be used by the cross-tools for a particular realtime system.

Test cases that are to be linked with the thread-safe version of the TCM and API must be compiled with `TET_POSIX_THREADS` defined, in order to make visible the threads-related contents of `tet_api.h`.

⁵ That is: a program that is linked with a child process controller and launched by a call to `tet_exec()` or `tet_spawn()`.

⁶ That is: a system that supports both multiple processes and a file system.

6.4.3 Supported API functions

6.4.3.1 Introduction

A substantial subset of the functions provided in the Lite version of the TET3 C API are implemented in TETware *professional* RT, as shown in the following subsections. The subsection headings correspond to those used in the chapters in the TETware Programmers Guide that describe the C API.

6.4.3.2 Test Case Structure and Management

The dynamic test case interface is not supported.
All the other interfaces are supported.

6.4.3.3 Insulating from the Test Environment

The configuration variables `TET_SIG_IGN` and `TET_SIG_LEAVE` apply to the Test Manager. The `TET3RT_SIG_IGN` and `TET3RT_SIG_LEAVE` variables can be set in the execute mode configuration to instruct the TCM/API to ignore signals and leave signals alone, respectively.

6.4.3.4 Error Handling and Reporting

All interfaces are supported.

6.4.3.5 Making Journal Entries

`tet_setcontext()` is supported but has no effect on the journal context number.
`tet_setblock()` causes the block number to be incremented in the Test Manager. All subsequent Test Case Information lines generated both by the Test Manager and the test case have the new block number.
All the other interfaces are supported.

6.4.3.6 Canceling Test Purposes

All interfaces are supported.

6.4.3.7 Accessing Configuration Variables

All interfaces are supported.

6.4.3.8 Generating and Executing Processes

`tet_fork()` and `tet_child` are supported only on profile 53 and 54 systems. In the thread-safe API, a call to `tet_fork()` creates a child process that contains only a copy of the calling thread.

The other interfaces are not supported.

6.4.3.9 Executed Process Functions

Sub-programs and `tet_main()` are not supported.

A call to `tet_exit()` calls the manufacturer-specific subsystem function `tet3rt_rt_exit()`. `tet_exit()` should only be called from a child process on profile 53 and 54 systems.

A call to `tet_logoff()` closes the communication channel to the Test Manager. This function should only be called from a child process on profile 53 and 54 systems, when API services are no longer required.

6.4.3.10 Test Case Synchronization

Not supported.

6.4.3.11 Remote System Information

Not supported.

6.4.3.12 Remote Process Control

Not supported.

6.4.3.13 Thread Functions

All the interfaces required to support POSIX threads are implemented in the thread-safe version of the API. These are: `tet_pthread_create()`, `tet_pthread_join()` and `tet_pthread_detach()`.

`tet_fork1()` is not supported.

Appendix A - TETware *professional* License

+++++++TET END USER LICENSE+++++++

BY DOWNLOADING THIS PRODUCT, YOU ARE CONSENTING TO BE BOUND BY THIS AGREEMENT. IF YOU DO NOT AGREE TO ALL OF THE TERMS OF THIS AGREEMENT, DO NOT INSTALL THE PRODUCT.

TETWARE PROFESSIONAL RELEASE 1 END USER LICENSE
REDISTRIBUTION NOT PERMITTED

This Agreement has two parts, applicable to the distributions as follows:

(A) Free binary evaluation copies - valid for up to 45 days, full functionality - no warranty,

(B) Licensed versions - full functionality, warranty fitness as described in documentation, includes annual support.

PART I (A above) -- TERMS APPLICABLE WHEN LICENSE FEES NOT (YET) PAID (LIMITED TO EVALUATION USE)

GRANT.

X/Open Company Limited, trading as The Open Group ('The Open Group') grants you a non-exclusive license to use the Software free of charge if your use of the Software is for the purpose of evaluating whether to purchase an ongoing license to the Software. The evaluation period for use by or on behalf of any entity is limited to a maximum of 90 days. If you are using the Software free of charge you are not entitled to hard-copy documentation, support or telephone assistance. If you fit within the description above, you may use the Software for any purpose and without fee.

DISCLAIMER OF WARRANTY.

Free of charge Software is provided on an 'AS IS' basis, without warranty of any kind.

THE OPEN GROUP DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT

SHALL THE OPEN GROUP BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

PART II (B above) -- TERMS APPLICABLE WHEN LICENSE FEES PAID

GRANT. Subject to payment of applicable license fees, The Open Group grants to you a non-exclusive license to use the Software and accompanying documentation ('Documentation') as described below.

Copyright (c) 1996, 1997, 1998, 1999, 2000 X/Open Company Ltd.

LIMITED WARRANTY.

The Open Group warrants that for a period of ninety (90) days from the date of acquisition, the Software, if operated as directed, will substantially achieve the functionality described in the Documentation. The Open Group does not warrant, however, that your use of the Software will be uninterrupted or that the operation of the Software will be error-free or secure.

EXCEPT AS STATED ABOVE THE OPEN GROUP DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE OPEN GROUP BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

SCOPE OF GRANT.

Permission to use for any purpose is hereby granted, as long as a support contract is in place.

Modification of the source is permitted.

Redistribution of the source code is not permitted without express written permission of The Open Group. Distribution of sources containing adaptations is expressly prohibited.

Redistribution of binaries or binary products containing TETware code is not permitted unless the distributor has a redistribution agreement with The Open Group.

Modifications sent to the authors are humbly accepted and it is their prerogative to make the modifications official.

TETware *professional* Real Time Guide

Portions of this work contain code derived from other versions of the Test Environment Toolkit, which are copyright

Copyright 1990,1992 Open Software Foundation

Copyright 1990,1992 Unix International

Copyright 1990,1992 X/Open Company Ltd.

Copyright 1991 Hewlett-Packard Co.

Copyright 1993 Information-Technology Promotion Agency, Japan

Copyright 1993 Sunsoft, Inc.

Copyright 1993 UNIX System Laboratories, Inc., a subsidiary of Novell Inc.

Copyright 1994,1995 UniSoft Ltd.

The unmodified source code of those works is freely available from <ftp.xopen.org>. The modified code contained in TETware *professional* restricts the usage of that code as per this license.

+++++

Appendix B - Manufacturer-specific Subsystems

B.1 Introduction

An interface has been defined which enables TETware *professional* RT components to send requests to other hardware and software subsystems. The implementation of the underlying functionality is specific to the hardware and/or software involved, and is implemented by (or on behalf of) the suppliers of these components. The following subsystems have been identified:

Communication subsystem Provides two-way communication between the Test Manager on the host system and the TETware *professional* RT TCM/API on the realtime system.

Exec subsystem Loads a test case executable on to the realtime system and executes it; and, provides a profile-independent mechanism for test case termination on the realtime system.

Reset subsystem Resets the realtime system.

Sections in this Appendix describe each subsystem in more detail, together with the API functions that must be provided by the user. In addition, some support functions provided by the Test Manager are described.

Functions that are implemented on the host system are used by the Test Manager, and functions that are implemented on the realtime system are used by the TETware *professional* RT version of the TCM and API library.

B.2 Subsystem Descriptions

B.2.1 Communication Subsystem

This subsystem consists of two parts; one part on the host system and the other on the realtime system. Each part is responsible for establishing a communication channel to the other part, and for exchanging fixed length message packets over the channel. Typically this subsystem is implemented using TCP/IP (if the realtime system supports it) or a connection between serial ports on each system.

B.2.2 Exec Subsystem

This subsystem consists of two parts; one part on the host system and the other on the realtime system.

The part on the host system is responsible for copying a program image file onto the realtime system and executing it. The part on the realtime system is used to terminate a running program, as if `exit()` has been called by the program.

B.2.3 Reset Subsystem

This subsystem consists of a single part on the host system. It is responsible for initializing the realtime system to a known state.

The following operations are defined:

- Soft reset;
- Hard reset.

Normally the Test Manager requests a soft reset if a test purpose times out, or if it is necessary to interrupt the currently running test purpose for some reason. If the soft reset operation fails, the Test Manager requests a hard reset. This process is analogous to sending a `SIGTERM` signal to a process running on a UNIX system, followed up by a `SIGKILL` signal if the process has not terminated within a reasonable time. If only one type of reset is possible for a particular realtime system, then it should be performed in response to both types of reset request.

B.3 Source File Directories

Source files for the API functions described here reside in the following directories:

`tet-root/src/tet3rt/msslib_native` Directory containing source files for subsystems that are implemented on the host system.

`tet-root/src/tet3rt/msslib_rtsys` Directory containing source files for subsystems that are implemented on the realtime system.

The TETware *professional* RT makefile scheme compiles the source files in these two directories using the appropriate compiler.

B.4 API Functions

The sections that follow define the interface to the manufacturer-specific subsystems for a particular combination of host system and realtime system.

An implementation of this API must be supplied for each realtime system that is to be used with TETware *professional* RT. The API is used by the Test Manager to deliver tests to the realtime system, provide services for the tests and receive results from the tests.

The API is designed to support different implementations of each facility. For example, the Communication subsystem might be implemented using a connection either over sockets or over a serial line.

The TETware *professional* RT distribution includes some example implementations of manufacturer-specific subsystems, which might be used as a starting point for a practical implementation. These are described in Sections B.12 to B.14 of this Appendix.

B.5 The `tet3rt.h` file

The file `tet-root/inc/tet3/tet3rt.h` contains declarations and definitions of all the interfaces that constitute this API.

In addition to the interfaces described here, the `tet3rt.h` file also contains declarations and definitions that are internal to TETware *professional* RT and are not part of this API. Users are reminded that only interfaces described in this chapter should be used by manufacturer-specific subsystem implementations.

B.6 Return Values

Each API call described here returns an integer whose value corresponds to one of the following manifest constants:

TET3RT_OK	Function succeeded.
TET3RT_EOF	A message operation encountered an End-of-File condition.
TET3RT_ER_ALREADY_OPEN	The communication channel is already open.
TET3RT_ER_CONFIG	Configuration variable error.
TET3RT_ER_EXEC	The program image could not be loaded on to the realtime system and/or executed.
TET3RT_ER_INVALID	Invalid parameter to function call.
TET3RT_ER_IO	An I/O error occurred.
TET3RT_ER_NOT_OPEN	The communication channel is not currently open.
TET3RT_ER_RESET	The realtime system could not be reset in the manner specified.
TET3RT_ER_RT_SYS_ID	Bad <code>rt_sys_id</code> argument to function call.
TET3RT_ER_SYSERROR	Error in system call (e.g., out of memory, can't fork, etc.)
TET3RT_ER_TIMEDOUT	Request timed out.

In the sections that follow, the possible return values for each function are included in the function's description.

B.7 Implementation Notes

B.7.1 Introduction

These notes are for guidance of implementers of APIs for all of the subsystems described here. Where appropriate, other implementation notes are included in the descriptions of the APIs for individual subsystems that follow.

B.7.2 Conditional Compilation

Conditional compilation may be used when it is necessary to provide different implementations of subsystems that run on the realtime system to support different POSIX realtime profiles. The TETware *professional* RT makefile scheme provides a compiler definition called `TET_POSIX13_PROFILE` whose value is set to the POSIX profile of the realtime system for which the TCM/API is being built. The following example illustrates how this definition might be used to determine whether or not multiple processes are supported on the realtime system:

```

#if TET_POSIX13_PROFILE == 51 || TET_POSIX13_PROFILE == 52
    /* single process profile */
#else
    /* multiple process profile */
#endif

```

B.7.3 Error Reporting

Each API function returns a value to indicate the success or failure of the requested operation. However, some of the error returns defined here can only convey fairly general information; particularly those that describe hardware or operating system errors.

When an API function is able to provide more information relating to the reason for a failure, it should do so by calling the TETware *professional* RT function `tet3rt_mss_printf()`. This function is described in Section B.11 “TETware *professional* RT Functions” later in this Appendix.

For example, suppose that the API function `tet3rt_msgtm_open()` is implemented using sockets. Such an implementation might typically make calls to `socket()`, `bind()` and `listen()`. If any of these calls fails, `tet3rt_msgtm_open()` would return `TET3RT_ER_IO`.

However, in order to provide more precise information in the event of failure of any of the underlying system calls, the API function should first make a call to `tet3rt_mss_printf()` to log a message stating the nature of the problem. Such a message would typically include a string describing the operating system error return, such as the string returned by a call to `strerror()`.

B.7.4 Use of Configuration Variables

A subsystem may choose to define configuration variables for its own use. In order to prevent name clashes, the names of variables used by manufacturer-specific subsystems should start with the prefix `TET3RT_MSS_`.

The value of a variable may be obtained by calling the TETware API function `tet_getvar()` on the host system. This function is described in the section entitled “Accessing configuration variables” in the TETware Programmers Guide.

Sometimes a subsystem may need to override the value of a parameter that the Test Manager passes to one of its functions. If necessary it can define a subsystem-specific configuration variable for this purpose.

For example: suppose a particular implementation of the Reset subsystem needed to know the name of a serial device to be used when performing a reset operation. In this case, the Reset subsystem might define an additional configuration parameter named (say) `TET3RT_MSS_RESET_TTY` and use the corresponding value when sending a reset request to the realtime system.

B.7.5 Signal Handling

Functions in this API should not ignore signals.

Functions in this API may block a signal for a short time in order to complete some atomic or time-critical operation. A signal should not be blocked for the duration of some operation, which might take some time to complete⁷. For example: a call to `select()` or `accept()`, calls to `read()` or `write()` on a slow device, etc. If a function in this API changes the disposition of a signal, the disposition should be restored. If a signal is received while a call to one of these API functions is in progress, the caller's signal handler may return to the caller via a call to `siglongjmp()`. This will cause the calling process to perform some cleanup operations (if possible) and exit.

The only "expected" signal that might cause this action to occur is the `SIGTERM` signal that might be delivered by TETware *professional* in response to a User Abort request. See Section 6.1.8 entitled "User Abort processing" earlier in this guide.

If a manufacturer-specific subsystem needs to perform additional cleanup operations on receipt of a `SIGTERM` signal, it should check the current disposition of the signal and, if it is not being ignored, save the current handler and install a new one. If subsequently receipt of a `SIGTERM` signal results in the subsystem-specific handler being called, the handler should:

1. Perform any required cleanup processing.
2. Restore the previous signal disposition that was saved when the subsystem-specific handler was installed.

⁷ Note that the message send and receive functions in the Communication subsystem do not need to concern themselves with signals, except to call `tet3rt_block_signals()` in the message receive function immediately before receiving a message packet. Further details are presented in the descriptions of `tet3rt_msgtm_rcv()` and `tet3rt_msgtm_send()` later in this Appendix.

3. Unblock SIGTERM, then send a SIGTERM to the current process. If some error results in the handler continuing execution after this point, the handler should print a suitable diagnostic and exit with a status of 1.

If a subsystem needs to perform this kind of processing, it should take care not to install its own signal handler more than once.

B.7.6 Trace Debugging

TETware *professional* RT includes a trace subsystem that can be used for debugging purposes. On the host system, the Test Manager implements this subsystem. On the realtime system, the TETware *professional* RT TCM/API implements this subsystem.

When an API function wishes to generate a trace message, it may do so by calling the TETware *professional* RT function `tet3rt_mss_trace()`. In addition, there are some other functions that may be useful when printing trace messages. All these functions are described in Section B.11 “TETware *professional* RT functions” later in this Appendix.

B.8 Communication Subsystem API

B.8.1 Introduction

The Communication subsystem is responsible for communications between the Test Manager on the host system and the TETware *professional* RT TCM/API on the realtime system. After the Test Manager has loaded and executed a test case on the realtime system, it opens a communication channel to the realtime system. At the same time the TCM/API on the realtime system opens the channel to the host system. Once the channel is open, the Test Manager and the TCM/API use it to exchange fixed length message packets. Each packet includes a magic number and a checksum so that TETware *professional* RT processes can readily detect communication errors. When the channel is no longer required, each side closes the channel. Typical implementations of this subsystem might use TCP/IP or serial port communications.

B.8.2 `tet3rt_msgtm_open()`

Synopsis

```
int tet3rt_msgtm_open(char *rtsys_id, int timeout);
```

Description

This function should be implemented on the host system. A call to this function opens a communication channel from the Test Manager to the realtime system. Since a Test Manager only tests a single realtime system, only one communication channel can be open at one time. The Test Manager calls this function after it has loaded and executed a test case on the realtime system.

Parameters

<code>rtsys_id</code>	Identifier for the realtime system to which this call applies. The format of this identifier is defined by the implementation. The Test Manager obtains the value for this parameter from the value of the <code>TET3RT_RTSYS_ID</code> variable in the execute mode configuration.
<code>timeout</code>	Specifies the number of seconds to wait for the open to complete.

Return Value

<code>TET3RT_OK</code>	The communication channel was opened successfully.
<code>TET3RT_ER_RTSYS_ID</code>	<code>rtsys_id</code> does not identify a known realtime system.
<code>TET3RT_ER_INVALID</code>	A parameter is invalid. This error should be returned if <code>rtsys_id</code> is <code>NULL</code> or points to an empty string.
<code>TET3RT_ER_IO</code>	An I/O error occurred while opening the communication channel.
<code>TET3RT_ER_SYSERROR</code>	A system error occurred (other than an I/O error).
<code>TET3RT_ER_ALREADY_OPEN</code>	The communication channel is already open.
<code>TET3RT_ER_CONFIG</code>	A required configuration variable is not defined or is set to an invalid value.
<code>TET3RT_ER_TIMEDOUT</code>	The communication channel could not be opened within the specified time.

Implementation Notes

If a socket is to be used for the communication channel, this call typically allocates a socket, listens for connections and accepts a connection when one arrives. If the subsystem uses a socket in passive mode, this leaves the realtime system free to choose whether or not to actually make the connection.

Once a channel has been opened, the subsystem should cache details of this channel for use in subsequent API calls to this subsystem.

B.8.3 tet3rt_msgtm_close()

Synopsis

```
int tet3rt_msgtm_close(void);
```

Description

This function should be implemented on the host system.

A call to this function closes the communication channel, which was opened by the last call to `tet3rt_msgtm_open()`.

Return Value

TET3RT_OK	The communication channel was closed successfully.
TET3RT_ER_NOT_OPEN	There is no currently open communication channel.
TET3RT_ER_IO	An I/O error occurred while closing the communication channel.
TET3RT_ER_SYSERROR	A system error occurred (other than an I/O error).

Implementation Notes

If there are requests waiting to be processed at the time of this call, they should be read and discarded before the call returns.

The subsystem may delete any cached information about the communication channel after a call to this function.

B.8.4 tet3rt_msgtm_recv()

Synopsis

```
int tet3rt_msgtm_recv(char *msgbuf, int timeout);
```

Description

This function should be implemented on the host system.

A call to this function reads a message packet of length `TET3RT_MSG_LEN` bytes from the realtime system, with timeout. This function returns when a packet has been received, or when the timeout expires.

Parameters

<code>msgbuf</code>	Pointer to a buffer of at least <code>TET3RT_MSG_LEN</code> bytes long, into which the implementation should put the received packet.
<code>timeout</code>	defines the number of seconds to wait for a packet to arrive. If no packet can be received within the specified number of seconds, the call should return a value of <code>TET3RT_ER_TIMEDOUT</code> . If <code>timeout</code> is zero, the call should return a packet if one is pending, otherwise it should return immediately.

Return Value

<code>TET3RT_OK</code>	A packet was received successfully.
<code>TET3RT_EOF</code>	EOF was encountered on the communication channel.
<code>TET3RT_ER_NOT_OPEN</code>	There is no currently open communication channel.
<code>TET3RT_ER_TIMEDOUT</code>	The timeout expired before a packet was received.
<code>TET3RT_ER_INVALID</code>	A parameter is invalid. This error should be returned if <code>msgbuf</code> is NULL; or, if <code>timeout</code> is negative.
<code>TET3RT_ER_IO</code>	An I/O error occurred on the communication channel.
<code>TET3RT_ER_SYSEERROR</code>	A system error occurred (other than an I/O error).

Implementation Notes

When the Test Manager is operating in server mode, it calls `tet3rt_msgtm_rcv()` with signals unblocked. The implementation should wait until a message packet is available for reading, then call the TETware *professional* RT function `tet3rt_block_signals()` immediately before reading the packet, so as to ensure that the read operation is not interrupted by a signal that is being caught by the Test Manager.

B.8.5 tet3rt_msgtm_send()**Synopsis**

```
int tet3rt_msgtm_send(char *msgbuf);
```

Description

This function should be implemented on the host system.

A call to this function sends a message packet of length `TET3RT_MSG_LEN` bytes to the realtime system.

Parameters

<code>msgbuf</code>	Pointer to a buffer containing the packet containing <code>TET3RT_MSG_LEN</code> bytes to be sent to the realtime system.
---------------------	---------------------------------------------------------------------------------------------------------------------------

Return Value

<code>TET3RT_OK</code>	The packet was sent successfully.
<code>TET3RT_ER_NOT_OPEN</code>	There is no currently open communication channel.
<code>TET3RT_ER_INVALID</code>	A parameter is invalid. This error should be returned if <code>msgbuf</code> is NULL.
<code>TET3RT_ER_IO</code>	An I/O error occurred on the communication channel.
<code>TET3RT_ER_SYSERROR</code>	A system error occurred (other than an I/O error).

B.8.6 tet3rt_msgrt_open()

Synopsis

```
int tet3rt_msgrt_open(void);
```

Description

This function should be implemented on the realtime system.

A call to this function opens a communication channel from the TCM on the realtime system to the Test Manager on the host system. Since a Test Manager only tests a single realtime system, only one communication channel can be open at one time.

The TCM calls this function soon after it starts executing.

Return Value

TET3RT_OK	The communication channel was opened successfully.
TET3RT_ER_IO	An I/O error occurred while opening the communication channel.
TET3RT_ER_SYSERROR	A system error occurred (other than an I/O error).
TET3RT_ER_ALREADY_OPEN	The communication channel is already open.

Implementation Notes

If a socket is to be used for the communication channel, this call typically allocates a socket and connects to the Test Manager on the host system.

Once a channel has been opened, the subsystem should cache details of this channel for use in subsequent API calls to this subsystem.

If calls to trace debugging functions are to be used in manufacturer-specific code on the realtime system, this function should set the `tet3rt_mss_traceflag` variable before it returns. Refer to the description of `tet3rt_mss_traceflag` later in this chapter for further details.

B.8.7 tet3rt_msgrt_close()

Synopsis

```
int tet3rt_msgrt_close(void);
```

Description

This function should be implemented on the realtime system.

A call to this function closes the communication channel, which was opened by the last call to `tet3rt_msgrt_open()`.

Return Value

TET3RT_OK	The communication channel was closed successfully.
TET3RT_ER_NOT_OPEN	There is no currently open communication channel.
TET3RT_ER_IO	An I/O error occurred while closing the communication channel.
TET3RT_ER_SYSERROR	A system error occurred (other than an I/O error).

Implementation Notes

If there are requests waiting to be processed at the time of this call, they should be read and discarded before the call returns. The subsystem may delete any cached information about the communication channel after a call to this function.

B.8.8 tet3rt_msgrt_send()

Synopsis

```
int tet3rt_msgrt_send(char *msgbuf);
```

Description

This function should be implemented on the realtime system.

A call to this function sends a message packet of length `TET3RT_MSG_LEN` bytes to the Test Manager on the host system.

Parameters

`msgbuf` Pointer to a buffer containing the packet containing `TET3RT_MSG_LEN` bytes to be sent to the Test Manager.

Return Value

<code>TET3RT_OK</code>	The packet was sent successfully.
<code>TET3RT_ER_NOT_OPEN</code>	There is no currently open communication channel.
<code>TET3RT_ER_INVALID</code>	A parameter is invalid. This error should be returned if <code>msgbuf</code> is <code>NULL</code> .
<code>TET3RT_ER_IO</code>	An I/O error occurred on the communication channel.
<code>TET3RT_ER_SYSERROR</code>	A system error occurred (other than an I/O error).

B.8.9 `tet3rt_msgrt_recv()`

Synopsis

```
int tet3rt_msgrt_recv(char *msgbuf, int timeout);
```

Description

This function should be implemented on the realtime system.

A call to this function reads a message packet of length `TET3RT_MSG_LEN` bytes from the Test Manager on the host system, with timeout. This function returns when a packet has been received, or when the timeout expires.

Parameters

<code>msgbuf</code>	Pointer to a buffer of at least <code>TET3RT_MSG_LEN</code> bytes long, into which the implementation should put the received packet.
<code>timeout</code>	Defines the number of seconds to wait for a packet to arrive. If no packet can be received within the specified number of seconds, the call should return a value of <code>TET3RT_ER_TIMEDOUT</code> . If <code>timeout</code> is zero, the call should return a packet if one is pending, otherwise it should return immediately.

Return Value

<code>TET3RT_OK</code>	A packet was received successfully.
<code>TET3RT_EOF</code>	EOF was encountered on the communication channel.
<code>TET3RT_ER_NOT_OPEN</code>	There is no currently open communication channel.
<code>TET3RT_ER_TIMEDOUT</code>	The timeout expired before a packet was received.
<code>TET3RT_ER_INVALID</code>	A parameter is invalid. This error should be returned if <code>msgbuf</code> is NULL; or, if <code>timeout</code> is negative.
<code>TET3RT_ER_IO</code>	An I/O error occurred on the communication channel.
<code>TET3RT_ER_SYSERROR</code>	A system error occurred (other than an I/O error).

Implementation Notes

When the TCM is operating in server mode, it calls `tet3rt_msgrt_recv()` with signals unblocked. The implementation should wait until a message packet is available for reading, then call the TETware *professional* RT function `tet3rt_block_signals()` immediately before reading the packet, so as to ensure that the read operation is not interrupted by a signal that is being caught by the TCM.

B.9 Exec Subsystem API

B.9.1 Introduction

The Exec subsystem provides support for executing processes on the realtime system.

B.9.2 tet3rt_rt_exec()

Synopsis

```
int tet3rt_rt_exec(char *rtsys_id, char *tcname);
```

Description

This function should be implemented on the host system. A call to this function copies a test case executable to the realtime system and executes it.

Parameters

<code>rtsys_id</code>	Identifier for the realtime system to which this call applies. The format of this identifier is defined by the implementation. The Test Manager obtains the value for this parameter from the value of the <code>TET3RT_RTSYS_ID</code> variable in the execute mode configuration.
<code>tcname</code>	Name of the file containing the test case to be copied to the realtime system.

Return Value

<code>TET3RT_OK</code>	The test case was copied and executed successfully.
<code>TET3RT_ER_RTSYS_ID</code>	<code>rtsys_id</code> does not identify a known realtime system.
<code>TET3RT_ER_INVALID</code>	A parameter is invalid. This error should be returned if one of the arguments is <code>NULL</code> or points to an empty string.

TET3RT_ER_CONFIG	A required configuration variable is not defined or is set to an invalid value.
TET3RT_ER_EXEC	An error occurred while the test case was being copied to, and/or executed on, the realtime system.

B.9.3 tet3rt_rt_exit()

Synopsis

```
int tet3rt_rt_exit(int status);
```

Description

This function should be implemented on the realtime system.

A call to this function terminates the calling process, as if by a call to `exit()`. This function is defined in order to provide the TETware *professional* RT TCM/API with a consistent way of terminating, irrespective of the POSIX profile supported by the realtime system. (This is because the `exit()` function is not specified for some of the POSIX realtime profiles.)

It is not necessary for this function to communicate the process exit status back to the Test Manager since, by the time the TCM/API calls this function, it has already done so.

Parameters

<code>status</code>	The process exit status. Whether or not the function can do anything useful with this value depends on the implementation.
---------------------	----------------------------------------------------------------------------------------------------------------------------

Return Value

This function must not return.

Implementation Notes

On profiles which only support a single process and where `exit()` or some equivalent function is not implemented, this function might simply go into an infinite loop. In this case, a subsequent call to `tet3rt_rt_exec()` should first perform an appropriate reset operation in order to gain control of the realtime system before loading and executing the test case.

B.10 Reset Subsystem API

B.10.1 Introduction

This subsystem provides the facility to reset the realtime system.

B.10.2 `tet3rt_rt_reset()`

Synopsis

```
int tet3rt_rt_reset(char *rtsys_id, int action);
```

Description

This function should be implemented on the host system.

Parameters

<code>rtsys_id</code>	Identifier for the realtime system to which this call applies. The format of this identifier is defined by the implementation. The Test Manager obtains the value for this parameter from the value of the <code>TET3RT_RTSYS_ID</code> variable in the execute mode configuration.
<code>action</code>	A value that indicates the type of reset operation to be performed (see below).

The following values are defined for `action`:

<code>TET3RT_SOFT_RESET</code>	Soft reset. The Test Manager uses this form of reset when a test purpose function times out or if execution must be interrupted for some other reason (e.g., on receipt of a User Abort instruction from <code>tcc</code>).
<code>TET3RT_HARD_RESET</code>	Hard reset.

The Test Manager uses this form of reset when a previous soft reset operation failed.

Return Value

TET3RT_OK	The request completed successfully. Note: it may not always be possible to determine whether or not the reset has been successful.
TET3RT_ER_RTSYS_ID	<code>rtsys_id</code> does not identify a known realtime system.
TET3RT_ER_INVALID	A parameter is invalid. This error should be returned if <code>rtsys_id</code> is NULL or points to an empty string.
TET3RT_ER_CONFIG	A required configuration variable is not defined or is set to an invalid value.
TET3RT_ER_RESET	The realtime system could not be reset in the manner specified.

B.11 TETware *professional* RT Functions

B.11.1 Introduction

These functions may be called from manufacturer-specific API functions. Except where indicated, they are provided both in the Test Manager on the host system, and in the TETware *professional* RT TCM/API on the realtime system.

These functions are declared in the file `tet3rt.h`.

B.11.2 Defined Constants

The following constants may be used in manufacturer-specific API functions. They are defined in the file `tet3rt.h`.

TET3RT_LNUMSZ	Maximum number of characters (including the sign and the terminating NULL) in the string representation of a long decimal value.
---------------	----------------------------------------------------------------------------------------------------------------------------------

TET3RT_LONUMSZ	Maximum number of characters (including the terminating NULL) in the string representation of a long octal value.
TET3RT_LXNUMSZ	Maximum number of characters (including the terminating NULL) in the string representation of a long hexadecimal value.

The file `<limits.h>` must be included before `tet3rt.h` in order to make these values visible.

B.11.3 Error Reporting

B.11.3.1 tet3rt_mss_printf()

Synopsis

```
void tet3rt_mss_printf(char *format, ...);
```

Description

This function may be called from a manufacturer-specific subsystem to report a detailed error message. Where possible, a message reported using this function is printed in the journal. If necessary, a long message may be divided into more than one line by including embedded new lines at suitable points.

Parameters

The parameters to this function are the same as for `printf()`.

Return Value

This function does not return a value.

B.11.3.2 tet3rt_mss_generror()

Synopsis

```
void tet3rt_mss_generror(int err, char *file, int line,  
char *s1, char *s2);
```

Description

This function is only provided in the Test Manager.

This function may be used to report a manufacturer-specific subsystem error message which consists of:

- the source file name and the line number where the error was detected;
- one or two user-supplied error message strings;
- a system error string obtained from a call to `strerror()`.

Typically it is invoked via a macro, which should be defined using the following code fragment:

```
/* error reporting */
static char srcFile[] = __FILE__;
#undef tet3rt_error          /* remove the definition in tet3rt.h */
#define tet3rt_error(err, s1, s2) \
    tet3rt_mss_genererror((err), srcFile, __LINE__, (s1), (s2))
```

When this is done, an error message may be generated using code similar to the following:

```
if (fopen(file, "r") == (FILE *) 0) {
    tet3rt_error(errno, "can't open", file);
    /* ... */
}
```

Parameters

<code>err</code>	The value of <code>errno</code> to be used when generating the system error message string to be appended to the message. If <code>err</code> is zero, no system error message string is generated.
<code>file</code>	The name of the source file to be used in the error message. Normally this is derived from the <code>__FILE__</code> macro that is defined by the C preprocessor.
<code>line</code>	The line number to be used in the error message. Normally this is derived from the <code>__LINE__</code> macro that is defined by the C preprocessor.
<code>s1</code>	The first part of the error message. This string must always be supplied.
<code>s2</code>	The second part of the error message. If no second part is to be printed, this parameter may be <code>NULL</code> .

Return Value

This function does not return a value.

B.11.4 Trace Debugging

B.11.4.1 tet3rt_mss_trace()

Synopsis

```
void tet3rt_mss_trace(int level, char *format, ...);
```

Description

This function enables a manufacture-specific subsystem to print a trace message using the trace debugging system in the calling process.

Parameters

level	Defines the trace level for this message, between 1 and 10. Generally speaking, a higher value should be used to indicate a greater level of verbosity.
format	The format to be used for the message, after the style of <code>printf()</code> .

Return Value

This function does not return a value.

B.11.4.2 tet3rt_mss_traceflag

Synopsis

```
extern int tet3rt_mss_traceflag;
```

Description

This variable is only implemented on the realtime system.

The trace system on the realtime system uses the value stored in this variable to decide whether or not MSS trace messages should be printed.

The Communication subsystem on the host system should obtain the value of the TET3RT_MSS_RT_SYS_TRACE variable in the execute mode configuration and send it to the Communication subsystem on the realtime system, which should then set tet3rt_mss_traceflag to this value. When this is done, the value specified by the configuration variable can be used to control the generation of manufacturer-specific subsystem trace messages on the realtime system.

B.11.4.3 tet3rt_mss_tdump()**Synopsis**

```
void tet3rt_mss_tdump(int level, char *buf, int len, char *title);
```

Description

This function enables a manufacturer-specific subsystem to request the trace debugging system in the calling process to print a hex dump of an area of memory.

Parameters

level	Defines the trace level for this memory dump, between 1 and 10. By convention, memory dumps are printed at trace level 10.
buf	Pointer to the first byte to be dumped.
len	Number of bytes to be dumped.
title	Text to be printed before the memory dump. If title is NULL, a default title is printed.

Return Value

This function does not return a value.

B.11.4.4 tet3rt_prmsser()

Synopsis

```
char *tet3rt_prmsser(int err);
```

Description

This function returns a printable representation of an API function return code. It may be used when constructing trace messages and other diagnostic strings.

Parameters

<code>err</code>	The API function return value whose symbolic value is to be printed.
------------------	----------------------------------------------------------------------

Return Value

Pointer to a string containing the symbolic value corresponding to `err`.

B.11.5 Signal Handling

B.11.5.1 tet3rt_block_signals()

Synopsis

```
void tet3rt_block_signals(void);
```

Description

A call to this function blocks signals that are being caught by the calling process.

The functions `tet3rt_msgtm_recv()` and `tet3rt_msgrt_recv()` should call this function after they have determined that a message packet is available for reading, but before the read operation starts. This should be done in order to ensure that the imminent read operation is not interrupted by signals that are being caught by the calling process.

Return Value

This function does not return a value.

B.12 Example MSS Implementations

Appendix C “Manufacturer-specific subsystems” describes the interfaces that must be implemented by the user for each realtime system on which test cases are to be executed by TETware *professional* RT. The TETware *professional* RT distribution contains some example MSS implementations that might be used as a starting point when customizing TETware *professional* RT to work with a particular realtime system.

In each example the Communication subsystem is a complete implementation, whereas the other subsystems are trivial implementations or only provide functionality sufficient for use during the TETware *professional* RT development process. Each example is described in the sections that follow.

B.13 Example Socket-based Implementation

B.13.1 Introduction

This MSS implementation might be suitable for use with a realtime system that supports TCP/IP. The source code for this example is in the following directories:

```
tet-root/src/tet3rt/msslib_native/socket_example
```

Test Manager components.

```
tet-root/src/tet3rt/msslib_rtsys/socket_example
```

Realtime system components.

B.13.2 Communication Subsystem

B.13.2.1 Subsystem Description

This subsystem uses sockets to communicate between the Test Manager on the host system and test cases running on a realtime system. The host name of the realtime system is specified by the `TET3RT_RTSYS_ID` variable in the execute mode configuration.

When the Test Manager calls `tet3rt_msgtm_open()`, the implementation acquires a stream socket and binds it to an ephemeral TCP port (the **listen port**). Then it forks a child process. This child process acquires a datagram socket and uses it to send configuration packets to a pre-defined UDP port (the **configuration port**) on the realtime system. The port number to use is hard-coded in the source code. Each configuration packet contains the host system's IP address and the port number of the listen port. At the same time the parent process listens for incoming connections on the listen port.

Meanwhile the test case on the realtime system acquires a datagram socket, binds it to the configuration port and waits for configuration packets to arrive from the Test Manager. When one arrives, the test case uses a stream socket to connect to the Test Manager using the IP address and port number specified in the configuration packet.

When the connection request arrives, the Test Manager accepts it in the parent, closes the listen socket and kills the child process. Then the connection is used to exchange message packets between the Test Manager on the host system and the test case on the realtime system.

B.13.2.2 Subsystem-specific Configuration Variables

This implementation of the Communication subsystem uses the following configuration variables:

`TET3RT_MSS_RTSYS_IP_ADDR`

This variable can be used to specify the IP address of the realtime system. When this variable is defined, the IP address specified is used instead of the one that would otherwise be derived from the value of `TET3RT_RTSYS_ID`.

Use of this variable is optional. It should only be defined when the realtime system's IP address cannot be derived from the value of `TET3RT_RTSYS_ID`.

`TET3RT_MSS_SERVER_IP_ADDR`

This variable can be used to specify the IP address that is to be used by test cases on the realtime system when connecting to the Test Manager on the Host system. When this variable

is defined, the IP address specified is used instead of the one that would otherwise be derived from the value of the host system's hostname.

Use of this variable is optional. It should only be defined when:

- the host system's hostname cannot be resolved to a single IP address, or
- the host system has more than one network interface and the IP address associated with the hostname refers to an interface other than the one to which the realtime system is connected.

TET3RT_MSS_RTSYS_TRACE

This variable specifies the trace level to be used in the MSS code on the realtime system. Use of this variable is optional; if not defined, its value defaults to zero.

B.13.3 Exec Subsystem

B.13.3.1 Test Manager Side

This is a simple implementation that uses `rcp` to copy a test case to the realtime system, and `rsh` to execute the test case on the realtime system. It is useful mainly when using a UNIX system to emulate the role of a realtime system. It is not suitable for use with an actual realtime system or in a production-testing environment.

B.13.3.2 Realtime System Side

This is a trivial implementation. On profile 51 and 52 systems, a call to `tet3rt_rt_exit()` goes into an infinite loop round a call to `pause()`. On profile 53 and 54 systems, a call to `tet3rt_rt_exit()` simply calls `exit()`.

If this implementation is used as the basis of a practical one, the code executed on profile 51 and 52 systems should be replaced by a call to a (platform-specific) process exit function, if there is one.

B.13.4 Reset Subsystem

This is a simple implementation that works in conjunction with the simple exec subsystem based on `rcp` and `rsh`. In this implementation, a call to `tet3rt_rt_reset()` simply sends a signal to a currently running `rsh` process. As with the simple exec subsystem described previously, this

implementation is useful mainly when using a UNIX system to emulate the role of a realtime system. It is not suitable for use with an actual realtime system or in a production-testing environment.

B.14 Example Serial Line Implementation

B.14.1 Introduction

This MSS implementation might be suitable for use with a realtime system which supports a serial port but which does not support TCP/IP. The source code for this example is in the following directories:

```
tet-root/src/tet3rt/msslib_native/serial_example
                                     Test Manager components.
tet-root/src/tet3rt/msslib_rtsys/serial_example
                                     Realtime system components.
```

B.14.2 Communication Subsystem

This subsystem uses a serial line to communicate between the Test Manager on the host system and test cases running on a realtime system. The name of the serial line to use on the host system is specified by the `TET3RT_RTSYS_ID` variable in the execute mode configuration. The name of the serial line to use on the realtime system, and other serial line parameters to use on both systems, are hard coded in the source files.

B.14.3 Exec Subsystem

B.14.3.1 Test Manager Side

This is a trivial implementation. A call to `tet3rt_rt_exec()` simply prompts the user to perform the required operations by hand. This implementation would not really be suitable for use in a production-testing environment.

B.14.3.2 Realtime System Side

The implementation of `tet3rt_rt_exit()` is the same as that provided in the example socket-based implementation.

B.14.4 Reset Subsystem

This is a trivial implementation. A call to `tet3rt_rt_reset()` simply prompts the user to perform the required operations by hand. This implementation would not really be suitable for use in a production-testing environment.